

Open|SpeedShop Reference Guide

September 30, 2018
Version 2.4.0

Contributions from Krell Institute, LANL, LLNL, SNL

Table of Contents

About this Manual	7
1 Introduction to Open SpeedShop.....	9
1.1 Basic Concepts, Interface, Workflow.....	9
1.1.1 Common Terminology	9
1.1.2 Concept of an Experiment	11
1.2 Performance Experiments Overview	11
1.2.1 Individual Experiment Descriptions	11
1.2.2 Synopsis of the Summary Experiment	13
1.2.3 Synopsis of the Sampling Experiments	13
1.2.4 Synopsis of the Tracing Experiments	14
1.2.5 Parallel Experiment Support	15
1.2.6 Vector instruction detection (AVX512 instruction detection)	15
1.3 Running an O SS Experiment.....	16
1.4 How to Gather and Understand Profiles	22
1.5 Description of the Granularity of views available in O SS	22
2.1 Overview/Summary (cbtfsummary) Experiment.....	24
2.1.1 Summary (cbtfsummary) experiment performance data gathering.....	24
2.1.2 Summary/Overview CSV directory structure and CSV file format definitions	25
2.1.3 Summary/Overview Report Generation	27
3.1 Program Counter Sampling (pcsamp) Experiment	28
3.1.1 Program Counter Sampling (pcsamp) experiment performance data gathering	29
3.1.1.1 Program Counter Sampling (pcsamp) experiment parameters	29
3.1.2 Viewing Program Counter Sampling (pcsamp) experiment performance data via GUI.....	29
3.1.3 Viewing Program Counter Sampling (pcsamp) experiment performance data via CLI.....	30
3.1.3.1 Vector Instruction view example (Intel based platforms only)	33
4.1 Call Path Profiling (usertime) Experiment.....	35
4.1.1 Call Path Profiling (usertime) experiment performance data gathering	35
4.1.2 Viewing Call Path Profiling (usertime) experiment performance data via GUI	36
4.1.3 Viewing Call Path Profiling (usertime) experiment performance data via CLI	39
4.1.4 Call Path Profiling (usertime) experiment function inline display.....	42
4.1.4.1 Call Path Profiling (usertime) experiment function inline display: Specific Kokkos Example:.....	46
4.1.4.2 Call Path Profiling (usertime) experiment function inline display: Specific Raja Example:.....	47
5 How to Relate Data to Architectural Properties	49
5.1 Hardware Counter Experiment (hwc)	50
5.1.1 Hardware Counter Threshold (hwc) experiment performance data gathering	51
5.1.2 Viewing Hardware Counter Threshold (hwc) experiment performance data via GUI.....	51
5.1.3 Viewing Hardware Counter Threshold (hwc) experiment performance data via CLI.....	53
5.2 Hardware Counter Time Experiment (hwctime)	54

5.2.1 Hardware Counter Time Threshold (hwctime) experiment performance data gathering	55
5.2.2 Viewing Hardware Counter Threshold (hwctime) experiment performance data via GUI	56
5.2.3 Viewing Hardware Counter Time Threshold (hwctime) experiment performance data via CLI	58
5.3 Hardware Counter Sampling (hwcsamp) Experiment.....	61
5.3.1 Hardware Counter Sampling (hwcsamp) experiment performance data gathering	63
5.3.1.1 Hardware Counter Sampling (hwcsamp) experiment parameters	63
5.3.2 Viewing Hardware Counter Sampling (hwcsamp) experiment performance data via GUI	63
5.3.2.1 Getting the PAPI counter as the GUI Source Annotation Metric	64
5.3.2.2 Viewing Hardware Counter Sampling Data via GUI.....	65
5.3.3 Viewing Hardware Counter Sampling (hwcsamp) experiment performance data via CLI	66
5.3.3.1 Job Script and osshwcsamp command.....	67
5.3.3.2 osshwcsamp experiment default CLI view.....	67
5.3.3.2 osshwcsamp experiment Status command and CLI view	69
5.3.3.3 osshwcsamp experiment Load Balance command and CLI view	69
5.3.3.4 osshwcsamp experiment Linked Object command and CLI view	69
5.3.3.5 osshwcsamp experiment displaying only the hwcsamp PAPI events CLI view	70
6 I/O Tracing and I/O Profiling	70
6.1 O SS I/O Tracing General Usage.....	70
6.2 I/O Base Tracing (io) experiment	70
6.2.1 I/O Base Tracing (io) experiment performance data gathering	71
6.2.2 Viewing I/O Base Tracing (io) experiment performance data via CLI	71
6.2.3 Viewing I/O Base Tracing (io) experiment performance data via GUI.....	71
6.3 I/O Extended Tracing (iot) experiment	71
6.3.1 I/O Extended Tracing (iot) experiment performance data gathering	71
6.3.2 Viewing I/O Extended Tracing (iot) experiment performance data via GUI.....	72
6.3.3 Viewing I/O Extended Tracing (iot) experiment performance data via CLI.....	74
6.4 I/O Lightweight Profiling (iop) General Usage	77
6.4.1 I/O Profiling (iop) experiment performance data gathering	77
6.4.2 Viewing I/O Profiling (iop) experiment performance data via GUI.....	77
6.4.3 Viewing I/O Profiling (iop) experiment performance data via CLI.....	79
7 Applying Experiments to Parallel Codes	81
8 MPI Tracing Experiments (mpi, mpit, mpip)	83
8.1 MPI Tracing Experiment (mpi)	92
8.1.1 MPI Tracing Experiment (mpi) performance data gathering	93
8.1.2 Viewing MPI Tracing Experiment (mpi) performance data via GUI.....	93
8.1.3 MPI Viewing Tracing Experiment (mpi) performance data via CLI.....	93
8.2 MPI Tracing Experiments (mpit).....	95
8.2.1 MPI Tracing Experiments (mpit) performance data gathering	95
8.2.2 Viewing MPI Tracing Experiments (mpit) performance data via GUI.....	96
8.2.3 Viewing MPI Tracing Experiments (mpit) performance data via CLI.....	96
8.3 MPI Tracing Experiments (mpip).....	98
8.3.1 MPI Tracing Experiments (mpip) performance data gathering	98
8.3.3 MPI Viewing Tracing Experiments (mpip) performance data via GUI.....	101

9 Threading Analysis Section	103
9.1 Threading Specific Experiment (pthreads).....	104
9.1.1 Threading Specific (pthreads) experiment performance data gathering (oss pthreads).....	105
9.1.2 Viewing Threading Specific (pthreads) experiment performance data via GUI.	106
9.1.3 Viewing Threading Specific (pthreads) experiment performance data via CLI..	107
9.2 OpenMP Related Performance Analysis.....	109
9.2.1 OpenMP Thread Wait Detection using OMPT interface.....	109
9.2.1.1 Augmentation of O SS sampling experiments	109
9.2.2 O SS OpenMP specific profiling experiment (omptp).....	113
9.2.2.1 OpenMP Specific (omptp) experiment performance data gathering (ossomptp)	113
9.2.2.2 Viewing OpenMP Specific (omptp) experiment performance data via GUI	113
9.2.2.3 Viewing OpenMP Specific (omptp) experiment performance data via CLI.....	113
9.3 Hybrid (OpenMP and MPI) Performance Analysis	114
9.3.1 Focus on individual Rank to get Load Balance for Underlying Threads	115
9.3.2 Clearing Focus on individual Rank to get back to default behavior	117
10 GPU Performance Analysis	119
10.1 NVIDIA CUDA Analysis Section	119
10.1.1 NVIDIA CUDA Tracing (cuda) experiment performance data gathering (oss cuda)	119
10.1.2 NVIDIA CUDA Experiment Performance Data Viewing using the new GUI.....	121
10.1.3 NVIDIA CUDA GUI Main Window User Interface Layout.....	121
10.1.4 Using the NVIDIA CUDA GUI to Analyze Application Performance.....	125
10.1.5 Viewing NVIDIA CUDA Tracing (cuda) experiment performance data via CLI	134
11 Memory Analysis Techniques.....	140
11.1 Memory Analysis Tracing (mem) experiment performance data gathering (oss mem).....	140
11.2 Viewing Memory Analysis Tracing (mem) experiment performance data via CLI	141
11.3 Viewing Memory Analysis Tracing (mem) experiment performance data via GUI	143
12 Advanced Analysis Techniques	146
12.1 Comparison Script Argument Description	147
12.1.1 osscompare metric argument.....	147
12.1.2 osscompare rows of output argument	148
12.1.3 osscompare output name argument.	148
12.1.4 osscompare view type or granularity argument	149
13 O SS User Interfaces	149
13.1 Command Line Interface Basics	150
13.1.2 CLI Metric Expressions and Derived Types	151
13.1.3 CLI Automatically Generated Derived Metrics and CLI Derived Metric Names	153
13.1.3.1 Computational Intensity.....	154
13.1.3.2 Level 1 Data Cache Miss Ratio	154
13.2 CLI Batch Scripting	154
13.3 Python Scripting	155
13.4 MPI_Pcontrol Support	155

13.5 Qt3 Legacy Graphical User Interface Basics	155
13.5.1 Basic Initial View – Default View	156
13.5.1.1 Icon ToolBar	156
13.5.1.2 View/Display Choice Selection.....	157
13.5.2 Preferences - How to change preferences	158
13.5.2.1 Disabling or enabling the preference for Save/Reuse views in CLI	161
13.6 Next Generation O SS GUI Application	163
13.6.1 Introduction	163
13.6.1.1 Main Window User Interface Layout	164
13.6.1.2 Case Studies of Using the O SS GUI to Analyze Experiment Results	176
14 Special System Support (Static Executables)	227
14.1 Cray and Blue Gene	227
14.1.1 osslink Command Information	228
14.1.2 Cray-Specific Static aprun Information	229
14.1.3 Changing parameters to the experiments	229
15 Setup and Build for O SS	231
15.1 Installing O SS with Spack.....	231
15.2 Installing O SS with the install-tool command	232
Build only the krell-root.....	232
Build cbtf components using the krell-root.....	232
Build only OSS using the cbtf components and the krell-root.....	232
15.3 Execution Runtime Environment Setup	233
15.3.1 Example module file	233
15.3.2 Example softenv file	236
15.3.3 Example dotkit file	237
16 Additional Information and Documentation Sources	238
16.1 Final Experiment Overview	238
16.2 Additional Documentation	238
17 Convenience Script Basic Usage Reference Information	240
17.1 Suggested Workflow.....	240
17.2 Convenience Scripts.....	240
17.3 Report and Database Creation.....	240
17.4 osscompare: Compare Database Files.....	241
17.5 osspcsamp: Program Counter Experiment	242
17.6 ossusertime: Call Path Experiment	242
17.7 osshwc, osshwctime: HWC Experiments	242
17.8 osshwcsamp: HWC Experiment.....	243
17.9 ossio, ossiot: I/O Experiments	243
17.10 ossmpi, ossmpip, ossmpit: MPI Experiments	244
17.11 ossmem: Memory Analysis Experiment	244
17.12 ossomptp: OpenMP Specific Profiling Experiment	245
17.13 osspthreads: POSIX Thread Analysis Experiment	245
17.14 osscuda: NVIDIA CUDA Tracing Experiment.....	245
17.15 cbtfsummary: Overview/Summary Multiple Metric Experiment	246
17.16 Key Environment Variables	246
Execution Related Variables	247
OPENSS_RAWDATA_DIR	247
OPENSS_ENABLE_MPI_PCONTROL	247

OPENSS_DATABASE_ONLY	247
OPENSS_RAWDATA_ONLY	247
OPENSS_DB_DIR	247
OPENSS_MPI_IMPLEMENTATION	248
OPENSS_DEFER_VIEW.....	248
CBTF_CSVDATA_DIR.....	248
Appendix A: cbtfsummary csv file format	249

About this Manual

Open|SpeedShop is an open-source multi-platform Linux performance tool to support performance analysis of applications running on both single-node and large-scale Intel, AMD, ARM, Intel Phi, PPC, Power and GPU processor-based systems and on Cray and IBM Blue Gene platforms.

This reference guide provides basic O|SS information. It's designed to help users understand the general O|SS experiments available to analyze application code. Extensive information is provided about how to employ these experiments and view performance information in practical ways, arming users to optimize and analyze their codes.

O|SS is a community effort with direct support from the Department of Energy National Nuclear Security Administration (DOE NNSA). It builds on a broad list of community infrastructures, most notably Dyninst and MRNet (Multicast Reduction Network) from the University of Wisconsin at Madison, the Libmonitor profiling tool from Rice University, and the Performance Application Programming Interface (PAPI) from the University of Tennessee at Knoxville.

O|SS is designed with usability in mind and is for application developers and computer scientists. The base functionality includes:

- High level Overview/Summary
- Program Counter Sampling
- Support for Call Stack Analysis
- Hardware Performance Counter Sampling and Threshold based
- MPI Lightweight Profiling and Tracing
- I/O Lightweight Profiling and Tracing
- Memory Trace Analysis
- OpenMP Profiling and Analysis
- POSIX Thread Trace Analysis
- NVIDIA CUDA Event Tracing and Hardware Counter Information

O|SS also is modular and extensible. It supports several levels of plugins, letting users add their own performance experiments.

The O|SS infrastructure and base components are released as open-source code primarily under LGPL. Highlights include:

- No need to recompile the user's application to get performance data at the function and library level. The debug option "-g" needed for statement, loop, and vector instruction level information.
- Comprehensive performance analysis for sequential, multithreaded and MPI applications

- Intel Only AVX512: Detection and display of vector instructions with 512 bit operands, showing address, opcode, time spent, and hardware maximum operand size for the vector instruction.
- Support for both first analysis steps and deeper analysis options for performance experts
- Easy-to-use GUI and fully scriptable through a command line interface and Python
- Supports Linux Systems and Clusters with Intel, AMD, ARM, and Power processors
- Extensible through new performance-analysis plugins, ensuring consistent look and feel
- In production use on all major cluster platforms at Los Alamos, Lawrence Livermore and Sandia national laboratories and at other sites around the world

Features include:

- Four user interface options: batch, command line, GUI and Python scripting API
- Supports multi-platform single-system image (SSI) and traditional clusters
- See the performance data in several levels of granularity:
 - Per library, per function, per loop, per statement and per vector instruction (only on Intel platforms – helps in AVX512 detection)
- Scales to large numbers of processes, threads and ranks
- Performance data viewable using multiple customizable means
- Performance experiment data and symbol information can be saved and restored for post-experiment analysis
- Performance data viewable for all of an application's lifetime or for smaller time slices
- Performance results comparable for processes, threads or ranks between a previous experiment and the current experiment
- GUI context-sensitive help.
- Interactive CLI help facility, which lists the CLI commands, syntax and typical usage
- Option to automatically group like-performing processes, threads or ranks

1 Introduction to Open|SpeedShop

Open|SpeedShop (O|SS) is an open-source performance analysis tool framework. It provides all the most common performance analysis steps in one tool via a common shared interface. It's easily extendable by writing plugins to collect and display performance data. It also comes with built-in experiments to gather and display several types of performance information.

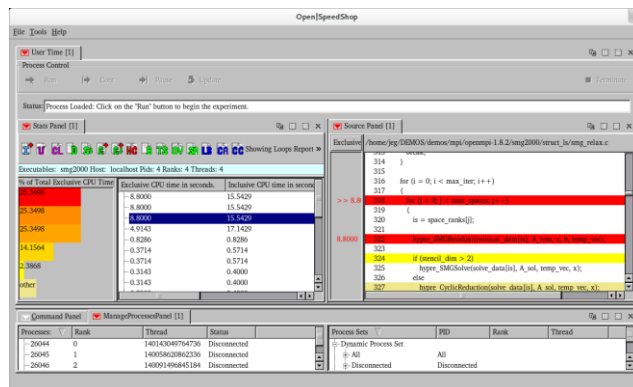
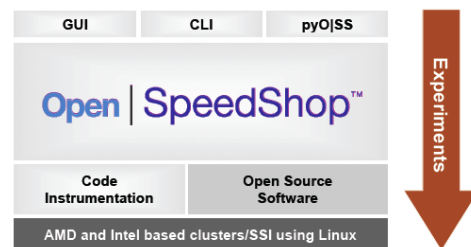
The existing O|SS experiments all work on unmodified application binaries. It has been tested on a variety of Linux clusters and supports Cray and IBM Blue Gene systems.

1.1 Basic Concepts, Interface, Workflow

Users can examine the results of O|SS performance tests, called experiments, in three ways: a GUI, a command line interface, or through Python libraries. Users also can apply these options to start experiments or start them by launching convenience scripts via the command line. For example, to commence a convenience script for the pcsamp experiment (Program Counter Sampling), the user executes the command:

```
osspsamp "<application>"
```

where <application> is the executable under study along with any arguments. The convenience scripts then will create a database of results from that experiment.



The user can examine any database in the GUI with the command:

```
openss -f <db file>
```

The GUI will provide simple graphics to help users understand the results and will relate the data back to the source code when possible.

1.1.1 Common Terminology

Technical terms can have multiple and/or context-sensitive meanings. This section explains and clarifies terms used in this document, especially with respect to O|SS tools.

Experiment: A set of collectors and an executable or executables joined to

generate performance information that viewable in human-readable form.

Focused Experiment: The current experiment that commands operate on. Users may run or view multiple experiments simultaneously, and unless a particular experiment is specified, the focused experiment will be used. Experiments are given enumerations, called experiment IDs, for identification.

Component(s): A component is a somewhat self-contained code section in the O|SS performance tool. This section does a set of specifically related tasks for the tool. For example, the GUI component does all the tasks related to displaying O|SS wizards, experiment creation, and results using a graphical user interface. The CLI component does similar functions but uses the interactive command-line delivery method.

Collector: The portion of the tool containing logic that gathers the performance metric. The collector part of the code is included in the experiment plugin.

Metric: The measurement the collector/experiment gathers. A metric could be a time, an occurrence counter or other property that reflects in some way on the application's performance and that a collector directly gathers during a performance experiment at application runtime.

Offline: An O|SS operating mode. This mode of operation uses a link override mechanism that lets performance data-gathering via libmonitor link O|SS performance data-gathering software components into the user application. For this operating mode, the application must be run from start-up to completion. The performance results may be viewed after the application terminates normally.

Param: User-set values that control the way each collector behaves. The parameter or **param** may cause the collector to perform various operations at certain time intervals or it may cause a collector to measure certain types of data. Although O|SS provides a standard way to set a parameter, it is up to the individual collector to decide what to do with that information. Documentation for each collector includes details about the available parameters.

Framework: The set of API functions that lets the user interface manage performance experiment creation and viewing. It connects the user interface and the cluster support and dynamic instrumentation components.

Plugin: A portion (library) of the performance tool that can be loaded and included in the tool at startup. Plugin development requires a tool-specific interface (API) so that it and the tool it's to go into can interact. Plugins normally are placed in a specific directory so tools know where to find them.

Target: The application or part of the application O|SS is running the experiment on. O|SS gives options that describe file names, host names, thread identifiers, rank identifiers and process identifiers, letting the user fine-tune what is targeted.

Granularity: O|SS gathers and displays data at five levels. The first four levels are most common and apply across all platforms. Those base levels are: per function, per statement, per loop, and per library. Recently, the vector instruction level was added for Intel platforms only. This allows users to tell what statements in their programs were vectorized, what the vector instructions are, the address of the vector instructions and the time spent executing the vector instruction.

1.1.2 Concept of an Experiment

In an O|SS experiment, a performance data collector gathers performance measurement data for a particular area of interest. The collector, which is a small dynamic or static object library, also contains functions that can interpret the gathered data into a human-understandable form. The experiment definition also includes the application under examination and how often the data will be collected (the sampling rate). The application's symbol information is saved into the experiment output file so that users can generate reports from the performance data file alone. The application itself need not be present to view the performance data at a later time.

1.2 Performance Experiments Overview

O|SS refers to the different performance measurements as experiments. Each experiment can measure and analyze different aspects of a code's performance. The user chooses the experiment type or type of data gathered. Any experiment may be applied to any application, except for applying MPI-specific experiments to non-MPI applications.

Each experiment consists of collectors and views. The collectors define specific performance data sources, such as program counter samples, call stack samples, hardware counters or library routine tracings. Views specify how the performance data is aggregated and presented to the user. It is possible to implement multiple collectors per experiment.

1.2.1 Individual Experiment Descriptions

The following table provides a quick overview of the experiment types that come with O|SS.

Experiment	Experiment Description
------------	------------------------

summary	Creates comma separated list (csv) files containing application level overview performance information on MPI, OpenMP, I/O, Memory usage, and hardware performance counters. Currently, this experiment is accessed via the cbtfsummary command.
pcsamp	Periodically samples the program counters, providing a low-overhead view of where time is spent in the user application.
usertime	Periodically samples the call path, letting the user view inclusive and exclusive time spent in application routines. It also lets the user see which routines called specific routines. Several views are available, including the “hot” path.
hwc	Counts hardware events (including clock cycles, graduated instructions, instruction and data cache, TLB misses and floating-point operations) at the machine instruction, source line and function levels.
hwcsamp	Similar to hwc, except sampling is based on time, not PAPI event overflows. Up to six events may be sampled during the same experiment.
hwctime	Similar to hwc, except it also includes call path sampling.
io	Accumulated wall-clock durations of input/output (I/O) system calls: read, readv, write, writev, open, close, dup, pipe, creat and others. Shows call paths for each unique I/O call path.
iop	Lightweight I/O profiling: Accumulated wall-clock durations of I/O system calls, including read, readv, write, writev, open, close, dup, pipe, creat and others, but doesn’t record individual call information.
iot	Similar to io, except it gathers more information, such as bytes moved, file names, etc.
mpi	Captures the time spent in and the number of times each MPI function is called. Shows call paths for each MPI unique call path.
mpip	Lightweight MPI profiling: Captures the time spent in and the number of times each MPI function is called. Shows call paths for each MPI unique call path, but doesn’t record individual call information.
mpit	Records each MPI function call event with specific data for display via a graphical user interface (GUI) or a command line interface (CLI). Trace format option displays the data for each call, showing its start and end times.
mem**	Tracks a potential memory allocation call that is not later destroyed (i.e., a leak). Records any memory allocation event that sets a new high-water mark for allocated memory current thread or process. Creates an event for each unique call path to a traced memory call and records the total number of times this call path was followed; the maximum allocation size, the minimum allocation size, and the total allocation; the total time spent in the call path;

	and the start time for the first call.
pthread	Captures the time spent in each POSIX thread function and the number of times each is called. Shows call paths for each POSIX thread function's unique call path.
omptp	Reports task idle, barrier, and barrier wait times per OpenMP thread and attributes those times to the OpenMP parallel regions.
cuda*	Captures the NVIDIA CUDA events that occur during the application execution and reports times spent for each event, along with the arguments for each event, in an event-by-event trace.

* Not presently available in OJSS offline mode.

**If run in offline mode, the memory experiment performance data is not reduced in the manner it is in the default mode because the filters are not called during offline mode.

1.2.2 Synopsis of the Summary Experiment

Currently, the summary experiment is accessed through a CBTF driver script named: `cbtfsummary`. The arguments to this script are similar to that of the OJSS convenience scripts, but the underlying infrastructure is somewhat different. This is explained in more detail in section 2.1.

The summary experiment gathers high-level information for a number of performance metrics, such as: MPI, OpenMP, Hardware counters, I/O, and Memory information. In the future, CUDA and sampling information may be added.

The summary experiment produces comma separated list (CSV) files of information for each thread of execution whose contents include application meta-data and performance information like timing and counts for the above mentioned metrics. The `CBTF_CSVDATA_DIR` environment variable can be used to set the directory path location for the `cbtfsummary` experiment csv files.

1.2.3 Synopsis of the Sampling Experiments

The program counter sampling (`pcsamp`), call path profiling (`usertime`), and hardware counter experiments (`hwc`, `hwctime`, `hwcsamp`) all use a form of sampling-based performance information-gathering techniques.

Program counter sampling (`pcsamp`) records the program counter (PC) in the specified user application by interrupting it at a user-defined time interval (with a default setting of 100 times a second at). This experiment provides a low-overhead overview of the application's time distribution. Its lightweight overview provides a good first step for analyzing an application's performance.

The call path profiling (usertime experiment) gathers the PC sampling information and records call stacks for each sample. This allows later display of application call path information and inclusive and exclusive timing data (see section 4.2). Use this experiment to find hot call paths (call paths that take the most time) and see who is calling whom.

The hardware counter experiments (hwc, hwctime, hwcsamp) access data like cache and TLB misses. The hwc and hwctime experiments sample hardware counter events based on an event threshold. The default event is PAPI_TOT_CYC overflows. (See chapter 5 for more information on PAPI and hardware counter-related experiments.) Instead of using a threshold, the hwcsamp experiment samples up to six events based on a sample time, similar to the usertime and pcsamp experiments. The hwcsamp experiment default events are PAPI_FP_OPS and PAPI_TOT_CYC.

1.2.4 Synopsis of the Tracing Experiments

Input/output tracing and profiling (io, iot, iop), MPI tracing (mpi, mpip, mpit), memory tracing (mem) and POSIX thread tracing (pthread) all use a form of tracing or wrapping of function names to record performance information. Tracing experiments do not use timers or thresholds to interrupt the application. Instead they intercept function calls of interest with a wrapper function that records timing and function argument information, calls the original function, and records this information for later viewing with OJSS's user interface tools.

The I/O tracing experiments (io, iot) record all POSIX I/O event invocations. They both provide aggregated and individual timings, while the iot experiment also provides argument information for each call. Use the I/O profiling experiment (iop) to get a more lightweight overview of application I/O usage. It records the invocation of all POSIX I/O events and accumulates the information, but does not save individual call information as the io and iot experiments do. That makes the iop experiment database smaller and the iop experiment faster than with the io and iot experiments.

The memory tracing experiment (mem) records invocation of all tracked memory function calls, also referred to as events. It provides aggregated and individual timings along with argument information for each call.

The MPI tracing experiments (mpi, mpit) record invocation of all MPI routines along with aggregated and individual timings. The mpit experiment also provides argument information for each call. Use the MPI profiling experiment (mpip) to get a more lightweight overview of application MPI usage. It records and accumulates the invocation of all MPI function call events, but does not save individual call information as the mpi and mpit experiments do. That makes the mpip experiment database smaller and the mpip experiment faster than with the mpi and mpit experiments.

The POSIX thread tracing experiment (pthreads) records invocation of all tracked POSIX thread-related function calls, also referred to as events. The pthreads experiment provides aggregated and individual timings and argument information for each call.

1.2.5 Parallel Experiment Support

O|SS supports MPI and threaded codes and it has been tested with various MPI implementations. Thread support is based on POSIX threads and OpenMP is supported in numerous ways, including via POSIX threads, OpenMP wait time sampling experiment augmentation and the omptp OpenMP profiling experiment.

Any O|SS experiment can be applied to any parallel application. This means you can run the program counter sampling experiment on a non-parallel application as well as an MPI or threaded application. Experiment data collectors are automatically applied to all tasks/threads. Default views aggregate (sum performance data) across all tasks/threads but data from individual tasks/threads are available. MPI calls are wrapped and MPI function elapsed time and parameter information is displayed.

1.2.6 Vector instruction detection (AVX512 instruction detection)

O|SS version 2.4.0 supports vector instruction detection to allow users to find what portions of their application are being executed with vector instructions. This feature is only supported on Intel processors.

The original effort was to detect Intel instructions that have 512 vector length arguments, i.e. AVX512 instructions. O|SS has broadened the original objective and now detects vector instructions in general and can report vector instructions of various vector lengths, where AVX512 is a special case of the general detection. This gives the O|SS tool flexibility to handle possible future architectural changes and report more categories for existing systems.

This functionality was implemented by using the gathered/sampled addresses to search through the executable during post processing the executable using the Dyninst API. The sampled addresses, which were gathered during the application execution, are matched with instruction at that address. The instruction must be one that Dyninst identifies as being in the vector category. The operands are examined for their size and the largest size is recorded and subsequently reporting in the O|SS command line interface tool (CLI). See section 1.5 *Description of the Granularity of views available in O|SS* for the view command information and section 3.1.3.1 *Vector Instruction view example (Intel based platforms only)* for additional information.

A convenience script option is required to gather this information, as statically scanning the application is necessary to detect the vector instruction information. The vector instructions will be detected if they are present on Intel based platforms.

1.3 Running an O|SS Experiment

First, consider what parameters you want to measure, then choose the appropriate experiment to run. You may want to start with the pcsamp experiment since it is lightweight and will give an overview of timing for the entire application. Once you have selected the experiment to run, you can launch it with either the wizard in the GUI or with command-line convenience scripts.

For example, say the user decides to run the pcsamp experiment on SMG2000, a semi-coarsening multigrid solver MPI application. On the command line, issue the command:

```
> osspcsamp "mpirun -np 256 smg2000 -n 60 60 60"
```

Where "mpirun -np 256 smg2000 -n 60 60 60" is a typical MPI command normally used to launch the smg2000 application. The MPI driver script or executable, mpirun, is used to launch SMG2000 on 256 processors and "-n 60 60 60" is passed as an argument to SMG2000.

Here's an example of an MPI SMG2000 pcsamp experiment run from a SLURM-based system using "srun" as the MPI driver, along with the application and experiment output:

```
> osspcsamp "srun -n 256 ./smg2000 -n 60 60 60"
```

```
[openss]: pcsamp experiment using the pcsamp experiment default sampling rate: "100".
[openss]: pcsamp experiment calling openss.
[openss]: Setting up offline raw data directory in /p/lscratchrb/fred/offline-oss
[openss]: Running offline pcsamp experiment using the command:
"srun -ppdebug -n 256 /collab/usr/global/tools/openspeedshop/oss-
dev/x8664/oss_offline_v2.1u6/bin/ossrun -c pcsamp ./smg2000 -n 60 60 60"
```

Running with these driver parameters:

```
(nx, ny, nz) = (60, 60, 60)
(Px, Py, Pz) = (256, 1, 1)
(bx, by, bz) = (1, 1, 1)
(cx, cy, cz) = (1.000000, 1.000000, 1.000000)
(n_pre, n_post) = (1, 1)
dim = 3
solver ID = 0
```

```
=====
Struct Interface:
```

```
=====
Struct Interface:
```

```
wall clock time = 0.020830 seconds
cpu clock time = 0.030000 seconds
```

```

=====
Setup phase times:
=====
SMG Setup:
wall clock time = 0.451188 seconds
cpu clock time = 0.460000 seconds
=====
Solve phase times:
=====
SMG Solve:
wall clock time = 2.707334 seconds
cpu clock time = 2.720000 seconds
Iterations = 7
Final Relative Residual Norm = 1.446921e-07
[openss]: Converting raw data from /p/lscratchrb/fred/offline-oss into temp file X.0.openss
Processing raw data for smg2000 ...
Processing processes and threads ...
Processing performance data ...
Processing symbols ...
Resolving symbols for /g/g24/fred/demos/workshop_demos/mpi/smg2000/test/smg2000
Resolving symbols for /lib64/ld-2.12.so
Resolving symbols for /collab/usr/global/tools/openspeedshop/oss-
dev/x8664/oss_offline_v2.1u6/lib64/openspeedshop/pcsamp-rt-offline.so
Resolving symbols for /collab/usr/global/tools/openspeedshop/oss-
dev/x8664/krellroot_v2.1u6/lib64/libmonitor.so.0.0.0
Resolving symbols for /usr/local/tools/mvapich-gnu-1.2/lib/shared/libmpich.so.1.0
Resolving symbols for /lib64/libc-2.12.so
Resolving symbols for /lib64/libpthread-2.12.so
Resolving symbols for /usr/lib64/libpsm_infinipath.so.1.14
Resolving symbols for /usr/lib64/libinfinipath.so.4.0
Updating database with symbols ...
Finished ...

[openss]: Restoring and displaying default view for:
/g/g24/fred/demos/workshop_demos/mpi/smg2000/test/smg2000-pcsamp.openss
[openss]: The restored experiment identifier is: -x 1

Exclusive % of CPU Function (defining location)
CPU time  Time
in
seconds.
272.1200 34.202 hypr_SMGResidual (smg2000: smg_residual.c,152)
195.0000 24.509 hypr_CyclicReduction (smg2000: cyclic_reduction.c,757)
80.0100 10.056 psm_mq_peek (libpsm_infinipath.so.1.14)
70.7600 8.893 ips_ptl_poll (libpsm_infinipath.so.1.14)
16.1300 2.027 hypr_SemiInterp (smg2000: semi_interp.c,126)
15.5600 1.955 __psmi_poll_internal (libpsm_infinipath.so.1.14)
14.2300 1.788 hypr_SemiRestrict (smg2000: semi_restrict.c,125)
6.5700 0.825 hypr_SMGAxpy (smg2000: smg_axpy.c,27)
6.0600 0.761 MPIR_Pack_Hvector (libmpich.so.1.0: dmpipk.c,31)
5.9500 0.747 ipath_dwordcpy (libinfinipath.so.4.0)
5.7900 0.727 MPID_DeviceCheck (libmpich.so.1.0: psmcheck.c,35)
...
...

```

When the application completes, a default report will be printed on screen. Performance information gathered during experiment execution will be stored in a database called smg2000-pcsamp.openss. Users can use the O|SS GUI to analyze the

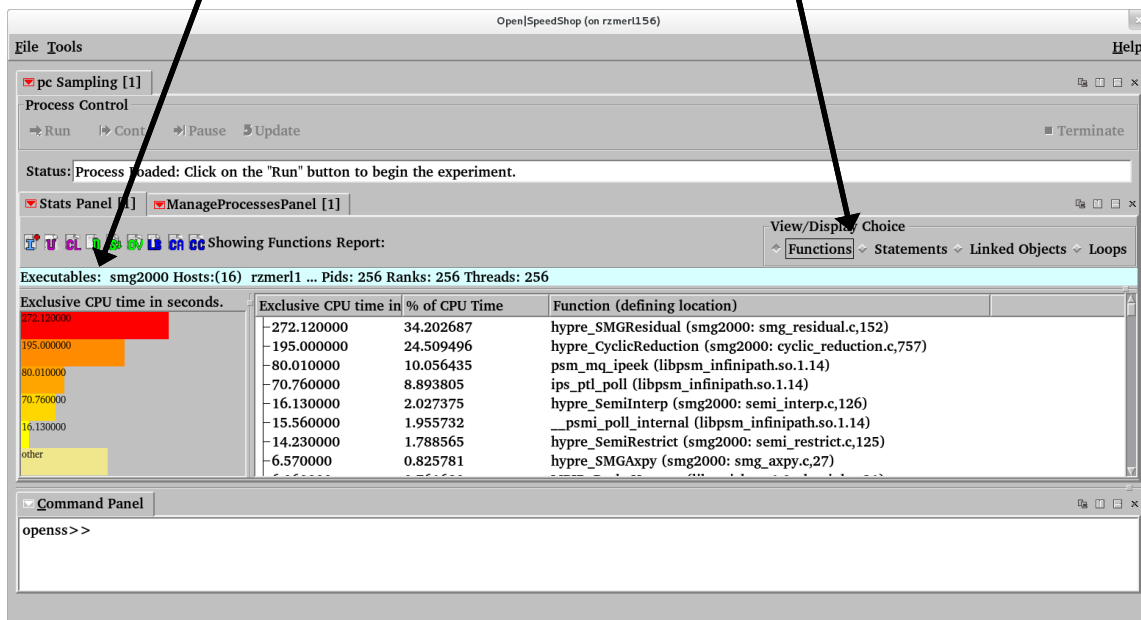
data in detail. Run the openss command to load that database file or open the file directly using the “-f” option:

```
> openss -f smg2000-pcsamp.openss
```

Here are basic examples of how to use the GUI to view the output database file created by the convenience script.

View Type Choices (D for Default selected and shown)
LB: Load Balance, CA: Cluster Analysis, and others available.

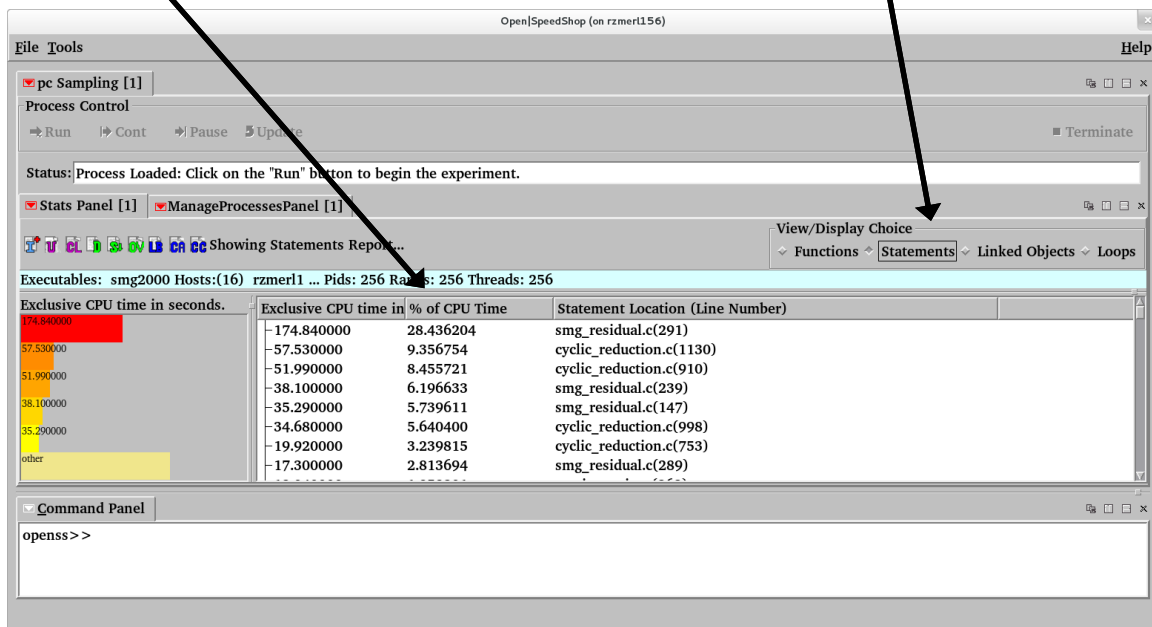
View Granularity Choices (Function level selected)
Statement, Library, and Loop level are available.



Users can choose to view data at Function, Statement, Linked Object, or Loop granularity levels. To switch from one view type to another, **first select the view granularity** (Function, Statement, Linked Object, or Loop), and **then select the type of view**. For the default views, select the “D” icon.

Statement in program that took the most time.

Statement Level Granularity Selected



Users can manipulate the windows within the GUI and double-click functions or statements to see the source code directly:

Double Click to open Source View and focus on source line (291)

Use window controls to split/arrange windows.
Vertical split was used here.

The screenshot shows the OpenSpeedShop (on rzmerl156) interface. The window is split vertically. The left pane contains the Stats Panel [1] and the Command Panel [1]. The right pane contains the Source Panel [1].

Stats Panel [1]

Exclusive CPU time in	% of CPU Time	Statement Location (Line Num)
174.840000	28.436204	smg_residual.c(291)
-57.530000	9.356754	cyclic_reduction.c(1130)
-51.990000	8.455721	cyclic_reduction.c(910)
-38.100000	6.196633	smg_residual.c(239)
-35.290000	5.739611	smg_residual.c(147)
-34.680000	5.640400	cyclic_reduction.c(998)
-19.920000	3.239815	cyclic_reduction.c(753)

Source Panel [1]

```
/g/g24/jeg/demos/workshop_demos/mpi/smg2000/struct_ls/smg_residual.c
284         x_data_box, start, base_stride, xi,
285         r_data_box, start, base_stride, ri);
286 #define HYPRE_BOX_SMP_PRIVATE loopk,loopi,loopj,Ai,xi,ri
287 #include "hypre_box_smp_forloop.h"
288
289     hypre_BoxLoop3For(loopi, loopj, loopk, Ai, xi, ri)
290     {
291         rp[ri] -= Ap[Ai] * xp[xi];
292     }
293     hypre_BoxLoop3End(Ai, xi, ri);
294 }
```

Command Panel [1]

Processes:	Rank	Thread	Status
-7537	224	46912523843552	Disconnected
-7538	225	46912523843552	Disconnected
-7539	226	46912523843552	Disconnected

Process Sets

PID	Rank	Thread
Dynamic Process Set		
All		
Disconnected	Disconnected	

1.4 How to Gather and Understand Profiles

A profile is the aggregated measurements collected during an experiment. Profiles examine code sections over time. They are advantageous because they reduce the size of performance data, which typically are collected with low overhead, providing a good overview of an application's operations.

The disadvantage of using a profile is that users must know beforehand how to aggregate the collected data. Since they provide more of an overview, profiles also omit performance details for individual events. There also could be an issue in which selecting an inappropriate sampling frequency skews profile results.

Statistical performance analysis is a standard profiling technique. It involves interrupting execution of the application at periodic intervals to record the execution (program counter value) location. It also can be used to collect additional data such as stack traces or hardware counters. Again, the advantage of this method is its low overhead. It is useful for getting an overview of the program and finding hotspots (time-intensive areas) within the program.

The sampling experiments available in O|SS include program counter sampling, call path profiling and hardware counter. The program counter sampling experiment (osspcsamp) provides approximate CPU time for each line and function in the program. The call path profiling experiment (ossusertime) provides inclusive vs. exclusive CPU time (see section 4.2) and includes call stacks. There are a number of Hardware Counter experiments (osshwc, osshwctime) that sample hardware counter overflows, plus osshwcsamp that can periodically sample up to six hardware counter events.

1.5 Description of the Granularity of views available in O|SS

Several base views are available for viewing of O|SS performance data through the command line interface (CLI) tool. The sampling experiments data lends itself to be viewed in more granularity than the tracing experiments due to the type of data metrics collected by O|SS. With sampling data, O|SS can display the performance information by library, by function, by loop, and by statement. Additionally, O|SS will gather information about vector instructions on Intel platforms only. That data is display per vector instruction.

Performance information gathered using tracing techniques is function based and therefore is only shown in per function granularity.

A summary of what to expect in O|SS performance views is as follows:

- Per library (linked object) (expview -vlinkedobjects in CLI)

- Counts, time spent are displayed on a per library or executable basis. This can give a good overview of the balance of MPI library time to base application time
- Per function (expview -vfunctions in CLI)
 - Counts, time spent, percentages are displayed on a per function basis. This allows the user to know which functions in the program are taking the most time, have the hardware counter hits, etc.
- Per loop (expview -vloops in CLI)
 - Display performance information based on loop granularity. Loops are determined via static binary analysis during the post process and performance information is attributed to the loop statements.
- Per statement (expview -vstatements in CLI)
 - Display performance information for each statement where O|SS collected information. This allows the user to know which statements in their program took the most time.
- Per vector instruction (expview -vvectorinstrs in CLI)
 - Intel platforms only
 - Display performance information based on vector instruction, showing the address, instruction opcode, and maximum hardware operand size.

2.1 Overview/Summary (cbtfsummary) Experiment

The summary experiment gathers high-level information for a number of performance metrics, such as:

- Time spent in MPI routines
- Time spent in OpenMP
 - Idle time
 - Barrier wait time
 - Barrier time
 - Implicit task time
 - Serial time
- Hardware counters
 - Cycles through a number of hardware counters and multiplexes
- Time spent in I/O
- Memory information
 - Dynamic memory size
 - Dynamic memory resident size
 - Dynamic memory high water mark
 - Dynamic memory shared size
 - Dynamic memory heap size
- In the future, CUDA, kokkos, and sampling information may be added.

Currently, the summary experiment is accessed through a CBTF driver script named: `cbtfsummary`. The arguments to this script are similar to that of the `OJSS` convenience scripts, but the underlying infrastructure is somewhat different. The `cbtfsummary` convenience script does not create a database files like the `oss<experiment name>` convenience script. `cbtfsummary` is intended to be highly scalable, therefore it produces comma separated list (CSV) files of information for each thread of execution whose contents include application meta-data and performance information like timing and counts for the above mentioned metrics.

The format of the csv files is described in section 2.1.2 and is subject to change as it is under development.

2.1.1 Summary (cbtfsummary) experiment performance data gathering

The summary experiment convenience script is “`cbtfsummary`”. Here’s how to use it to gather overview information about your application run.

```
cbtfsummary “how you normally run your application”
```

An example of a summary experiment run on the `nbody` application is as follows:

```
> setenv CBTF_CSVDATA_DIR ./sierra_csvdata  
> cbtfsummary “jsrun -n 16 ./nbody”
```

The first line defines where the csv files will be written. The second line runs the application, nbody in this case. cbtfsummary gathers the performance information and writes the csv files, one for each thread of execution to the directory specified by **CBTF_CSVDATA_DIR**. If **CBTF_CSVDATA_DIR** is not specified, then the files are written to /tmp.

2.1.2 Summary/Overview CSV directory structure and CSV file format definitions

Currently, the cbtfsummary experiment provides high level performance information related to MPI, OpenMP, POSIX I/O, POSIX memory, and hardware counters. See Appendix A for a detailed of the description of the cvs file contents for the cbtfsummary experiment. As cbtfsummary is in active development there are possible changes are coming for the cbtfsummary functionality. User feedback is appreciated.

There is a debug option that is recognized by the cbtfsummary which causes the tool to show what the summary collector is writing to the csv files. For illustration, the example run below shows what is available.

Example run:

```
$ CBTF_SHOW_CSVDATA=1 cbtfsummary "mpirun -np 2 ./nbody-openmpi"
Iteration 50 of 50...
[17018,0] host,pid,rank,tid,posix_tid,executable,total_time_seconds
[17018,0] localhost.localdomain,17018,1,0,140353435236160,nbody-openmpi,3.650807
[17018,0] maxrss_bytes,utime_seconds,stime_seconds
[17018,0] 8348,3.377000,0.083000
[17018,0] dmem_size,dmem_resident,dmem_high_water_mark,dmem_shared,dmem_heap
[17018,0] 127356,7844,8348,4516,76136
[17018,0] allocation_time,allocation_calls,allocation_bytes
[17018,0] 0.000050,65,181172
[17018,0] free_time,free_calls
[17018,0] 0.000036,61
[17018,0] total_mpi_time
[17018,0] 0.368264
[17018,0]
PAPI_TOT_CYC,PAPI_TOT_INS,PAPI_LD_INS,PAPI_L3_TCM,PAPI_L2_TCM,PAPI_L1_TCM,PAPI_TLB_IM,
PAPI_REF_CYC,PAPI_FUL_CCY,PAPI_RES_STL
[17018,0]
17533265404,38594596124,16423184578,161232,816220,2925356,26860,649765196,915336617
8,6696100928
[17017,0] host,pid,rank,tid,posix_tid,executable,total_time_seconds
[17017,0] localhost.localdomain,17017,0,0,140684808259392,nbody-openmpi,3.651217
[17017,0] maxrss_bytes,utime_seconds,stime_seconds
[17017,0] 8316,3.375000,0.082000
[17017,0] dmem_size,dmem_resident,dmem_high_water_mark,dmem_shared,dmem_heap
[17017,0] 127356,7840,8316,4520,76136
[17017,0] allocation_time,allocation_calls,allocation_bytes
[17017,0] 0.000037,67,341172
[17017,0] free_time,free_calls
[17017,0] 0.000027,63
```

```
[17017,0] total_mpi_time
[17017,0] 0.340121
[17017,0]
PAPI_TOT_CYC,PAPI_TOT_INS,PAPI_LD_INS,PAPI_L3_TCM,PAPI_L2_TCM,PAPI_L1_TCM,PAPI_TLB_IM,
PAPI_REF_CYC,PAPI_FUL_CCY,PAPI_RES_STL
[17017,0]
17532945468,38099297260,16219346728,129228,1072288,2938000,23824,649522092,90207017
90,6668198566
All Threads are finished.
```

cbtfsuammary creates these named folders in the directory specified by CBTF_CSVDATA_DIR or in the default /tmp location. Each time cbtfsuammary is run a new directory is created with an increasing integer appended so that the previous runs data is not destroyed.

```
$ ls -ld ./nbody-openmpi-overview-csvdata-*
drwxrwxr-x. 4 fred fred 4096 Jun 13 16:51 ./nbody-openmpi-overview-csvdata-0
drwxrwxr-x. 4 fred fred 4096 Jun 13 23:14 ./nbody-openmpi-overview-csvdata-1
drwxrwxr-x. 4 fred fred 4096 Jun 13 23:30 ./nbody-openmpi-overview-csvdata-2
drwxrwxr-x. 4 fred fred 4096 Jun 13 23:30 ./nbody-openmpi-overview-csvdata-3
drwxrwxr-x. 4 fred fred 4096 Jun 14 13:16 ./nbody-openmpi-overview-csvdata-4
```

Each of the above directories represents one run of cbtfsuammary. To view the cvs files from one run, examine the directory of interest.

```
$ ls -latr nbody-openmpi-overview-csvdata-4
total 48
drwxrwxr-x. 2 fred fred 4096 Jun 14 13:16 localhost.localdomain-1
drwxrwxr-x. 2 fred fred 4096 Jun 14 13:16 localhost.localdomain-0
drwxrwxr-x. 4 fred fred 4096 Jun 14 13:16 .
drwxrwxr-x. 24 fred fred 32768 Jun 14 13:22 ..
```

```
$ ls -latr nbody-openmpi-overview-csvdata-4/*
nbody-openmpi-overview-csvdata-4/localhost.localdomain-1:
total 12
-rw-rw-r--. 1 fred fred 635 Jun 14 13:16 nbody-openmpi-1-0.csv
nbody-openmpi-overview-csvdata-4/localhost.localdomain-0:
total 12
-rw-rw-r--. 1 fred fred 636 Jun 14 13:16 nbody-openmpi-0-0.csv
```

A cat of csv files shows the contents of the csv files.

```
$ cat nbody-openmpi-overview-csvdata-4/*/*csv
host,pid,rank,tid,posix_tid,executable,total_time_seconds
localhost.localdomain,17017,0,0,140684808259392,nbody-openmpi,3.651217
maxrss_bytes,utime_seconds,stime_seconds
8316,3.375000,0.082000
dmem_size,dmem_resident,dmem_high_water_mark,dmem_shared,dmem_heap
127356,7840,8316,4520,76136
allocation_time,allocation_calls,allocation_bytes
0.000037,67,341172
free_time,free_calls
0.000027,63
```

```

total_mpi_time
0.340121
PAPI_TOT_CYC,PAPI_TOT_INS,PAPI_LD_INS,PAPI_L3_TCM,PAPI_L2_TCM,PAPI_L1_TCM,PAPI_TLB_IM,
PAPI_REF_CYC,PAPI_FUL_CCY,PAPI_RES_STL
17532945468,38099297260,16219346728,129228,1072288,2938000,23824,649522092,90207017
90,6668198566
host,pid,rank,tid,posix_tid,executable,total_time_seconds
localhost.localdomain,17018,1,0,140353435236160,nbody-openmpi,3.650807
maxrss_bytes,utime_seconds,stime_seconds
8348,3.377000,0.083000
dmem_size,dmem_resident,dmem_high_water_mark,dmem_shared,dmem_heap
127356,7844,8348,4516,76136
allocation_time,allocation_calls,allocation_bytes
0.000050,65,181172
free_time,free_calls
0.000036,61
total_mpi_time
0.368264
PAPI_TOT_CYC,PAPI_TOT_INS,PAPI_LD_INS,PAPI_L3_TCM,PAPI_L2_TCM,PAPI_L1_TCM,PAPI_TLB_IM,
PAPI_REF_CYC,PAPI_FUL_CCY,PAPI_RES_STL
17533265404,38594596124,16423184578,161232,816220,2925356,26860,649765196,915336617
8,6696100928

```

2.1.3 Summary/Overview Report Generation

After cbtfsummary experiment completes, a human readable, formatted report is created containing the minimum, maximum, and average values for the data metrics described above, in section 2.1.2. The report will be displayed to standard out (stdout) and a file will be created with the identical information and file will be placed in the directory from which the cbtfsummary experiment was run.

The following example and report is from a cbtfsummary experiment on the nbody application using 72 ranks. The commands used were:

```

export CBTF_CSVDATA_DIR=/lustre/scratchb/Fredrick
cbtfsummary "mpirun -np 72 ./nbody"

```

The cbtfsummary experiment created csv files for each of the rank and wrote the files into the directory path: /lustre/scratchb/fredrick. This directory structure and csv files can be mined by other tools, if desired.

The cbtfsummary report from the example run is shown below.

Metrics for 0 72

metric name	max	min	avg
dmem_high_water_mark	41832	43680	43779
dmem_resident	19588	12992	13452
dmem_size	224712	294248	306507
dmem_shared	7188	7240	7548
dmem_heap	78508	71836	71989
read_time	0.000000	0.000000	0.000000
read_bytes	0	0	0
io_total_time	0.000051	0.000011	0.000021
write_bytes	0	0	0
write_time	0.000000	0.000000	0.000000
allocation_calls	173	93	125
allocation_time	0.000138	0.000042	0.000067
allocation_bytes	1056651	12803	60898
free_calls	98	74	76
free_time	0.000011	0.000005	0.000009
total_mpi_time	9.929802	9.620426	9.748008
PAPI_DP_OPS	152486080	146049884	1268564188
PAPI_TOT_INS	1097995836	1083669158	557419126
PAPI_VEC_DP	152486080	146049884	149964029
PAPI_TOT_CYC	878111892	869962518	876642689
PAPI_LD_INS	503700340	491832406	149964029
maxrss_bytes	44104	37516	43779
stime_seconds	1.184613	0.111020	1.354460
utime_seconds	0.372336	0.377667	0.363017
total_time_seconds	9.996704	10.016452	9.885804

3.1 Program Counter Sampling (pcsamp) Experiment

A flat profile will answer the basic question, “Where does my code spend its time?” O|SS displays this as a list of code elements of varying granularity – statements, functions and libraries (linked objects) – with the time spent at each function. Flat profiling can be done through sampling, letting the user avoid the overhead of direct measurements. But users must request a sufficient number of samples (sampling rate) to get an accurate result.

The profile displays the time spent per function or per statement, helping identify critical, computationally intensive code regions. While viewing this, the user must ask:

- Are those the functions/statements that were expected relative to consuming the most time?
- Does this match the computational kernels?
- Are any runtime functions consuming a lot of time?

The goal is to identify components that are bottlenecks. To do this, view the profile aggregated by shared (linked) objects, ensuring the correct or expected modules are present, then analyze the impact of those support and/or runtime libraries.

3.1.1 Program Counter Sampling (pcsamp) experiment performance data gathering

The program counter sampling experiment convenience script is “osspsamp”. Here’s how to use it to gather address values in which O|SS periodically interrupted the application and took an address sample:

```
osspsamp “how you normally run your application” < sampling rate>
```

An example of flat profiling would be to run the program counter sampling in O|SS. We will run the convenience script on our test program, SMG2000:

```
> osspsamp “mpirun -np 256 smg2000 -n 50 50 50”
```

It is recommended that users compile their code with the `-g` option to see the statements in the sampling. Sampling frequency is an optional parameter, with settings of high (200 samples per second), low (50 samples per second) and default (100 samples per second). To run the same experiment with the high sampling rate, issue the command:

```
> osspsamp “mpirun -np 256 smg2000 -n 50 50 50” high
```

3.1.1.1 Program Counter Sampling (pcsamp) experiment parameters

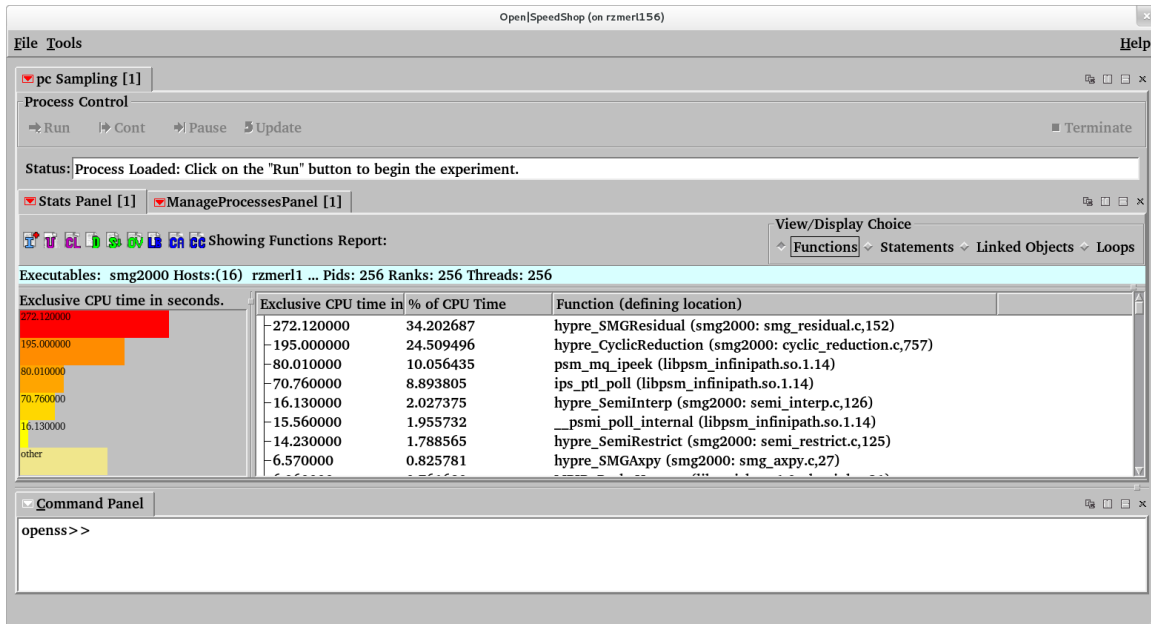
The pcsamp experiment is timer-based: A timer periodically interrupts the processor and the address in the program counter is read and saved each time. This allows O|SS to map those address values back to the source when a user views the pcsamp performance information via the CLI or GUI tool.

In the next example, the user chooses to sample only 45 times a second instead of the default 100. One reason for this would be to save database size; a lower sampling rate may still give an accurate portrayal of application behavior.

```
osspsamp “how you normally run your application” 45
```

3.1.2 Viewing Program Counter Sampling (pcsamp) experiment performance data via GUI

To view results of this flat profile in the O|SS GUI, use the “openss -f <database filename>” command.



3.1.3 Viewing Program Counter Sampling (pcsamp) experiment performance data via CLI

After running a program counter experiment via the command:

```
osspsamp "mpirun -np 4 ./smg2000 -n 65 65 65"
```

the user can use the following command to open the newly created database file and view the data in the CLI:

```
openss -cli -f smg2000-pcsamp-0.openss
```

Once inside the CLI, several commands can be used to view this performance information. Here are some examples:

For the default view, use the `expview` command with no arguments.

```
openss>>expview
```

```
Exclusive  % of CPU Function (defining location)
CPU time   Time
  in
seconds.
7.640000 41.657579 hypre_SMGResidual (smg2000: smg_residual.c,152)
4.840000 26.390403 hypre_CyclicReduction (smg2000: cyclic_reduction.c,757)
0.800000 4.362050 mca_btl_vader_check_fboxes (libmpi.so.12.0.2: btl_vader_fbox.h,184)
0.450000 2.453653 unpack_predefined_data (libopen-pal.so.13.0.2: opal_datatype_unpack.h,34)
0.400000 2.181025 hypre_SemiInterp (smg2000: semi_interp.c,126)
0.370000 2.017448 __memcpy_ssse3_back (libc-2.17.so)
0.350000 1.908397 pack_predefined_data (libopen-pal.so.13.0.2: opal_datatype_pack.h,35)
0.330000 1.799346 hypre_SemiRestrict (smg2000: semi_restrict.c,125)
0.310000 1.690294 opal_progress (libopen-pal.so.13.0.2: opal_progress.c,151)
0.180000 0.981461 opal_sys_timer_get_cycles (libopen-pal.so.13.0.2: timer.h,43)
```

...
...
...

To view performance information by statement granularity, use the `-v statements` argument to `expview`. The top of the resulting list shows the statement in the application that took the most time for this particular SMG2000 run. Results may vary when running on other platforms and/or with different SMG2000 arguments or different compiler options.

```
openss>>expview -v statements
```

Exclusive CPU time	% of CPU Time	Statement Location (Line Number)
in seconds.		
5.790000	35.304878	smg_residual.c(289)
1.410000	8.597561	cyclic_reduction.c(1130)
1.080000	6.585366	smg_residual.c(238)
0.830000	5.060976	cyclic_reduction.c(910)
0.690000	4.207317	cyclic_reduction.c(999)
0.420000	2.560976	cyclic_reduction.c(1061)
0.410000	2.500000	smg_residual.c(287)
0.330000	2.012195	cyclic_reduction.c(853)
0.260000	1.585366	opal_datatype_unpack.h(59)
0.260000	1.585366	cyclic_reduction.c(1000)
0.240000	1.463415	btl_vader_fbox.h(197)
0.230000	1.402439	semi_restrict.c(262)
0.200000	1.219512	opal_datatype_pack.h(60)
0.180000	1.097561	cyclic_reduction.c(1131)
0.150000	0.914634	semi_interp.c(294)
...		
...		

Use the `-v linkedobjects` argument to `expview` to see performance data for SMG2000 at the library (linked object) granularity. This tells how much time was spent in each of the libraries from which OJSS took program counter samples.

This view gives strictly an overview of where time was spent from the library perspective. If the MPI library time is very high, it may indicate that this run was using MPI ineffectively. This may indicate load imbalance.

```
openss>>expview -v linkedobjects
```

Exclusive CPU time	% of CPU Time	LinkedObject
in seconds.		
14.180000	76.981542	smg2000
1.860000	10.097720	libmpi.so.12.0.2
1.630000	8.849077	libopen-pal.so.13.0.2
0.740000	4.017372	libc-2.17.so
0.010000	0.054289	ld-2.17.so

```
openss>>
```

Users also can apply `-v` loops as an argument to view time spent at the loop level of granularity. For example, the first line in the display shows that a loop starting at line 204 in `smg_residual.c` was the one consuming the most time. O|SS cannot accurately determine the loops end statement, so only the starting line number is displayed.

```
openss>>expview -v loops
```

```
Exclusive % of CPU Loop Start Location (Line Number)
CPU time   Time
in
seconds.
7.640000 32.345470 smg_residual.c(204)
2.240000 9.483489 cyclic_reduction.c(1022)
2.140000 9.060119 cyclic_reduction.c(882)
0.790000 3.344623 btl_vader_fbox.h(188)
0.550000 2.328535 cyclic_reduction.c(1034)
0.430000 1.820491 cyclic_reduction.c(851)
0.430000 1.820491 cyclic_reduction.c(851)
0.430000 1.820491 cyclic_reduction.c(851)
0.430000 1.820491 cyclic_reduction.c(835)
0.410000 1.735817 opal_datatype_unpack.h(58)
...
...
```

Another useful CLI command is `expstatus`, which provides a summary of metadata for the O|SS experiment. This command displays the experiment's metadata. What host it was run on, the time of the experiment, number and details about the MPI ranks, threads, and/or processes involved in the experiment.

```
openss>>expstatus
```

```
Experiment definition
{ # ExpId is 1, Status is Terminated, Saved database is smg2000-pcsamp-0.openss
  Performance data spans 4.773728 seconds from 2016/11/22 07:43:30 to 2016/11/22 07:43:35
  Executables Involved:
    smg2000
  Currently Specified Components:
    -h localhost -p 8090 -t 0 -r 1 (smg2000)
    -h localhost -p 8091 -t 0 -r 2 (smg2000)
    -h localhost -p 8092 -t 0 -r 3 (smg2000)
    -h localhost -p 8089 -t 0 -r 0 (smg2000)
  Previously Used Data Collectors:
    pcsamp
  Metrics:
    pcsamp::percent
    pcsamp::threadAverage
    pcsamp::threadMax
    pcsamp::threadMin
    pcsamp::time
```

Parameter Values:
pcsamp::sampling_rate = 100
Available Views:
pcsamp

3.1.3.1 Vector Instruction view example (Intel based platforms only)

Show the vector instruction detection data gathered by O|SS in a CLI optional view. The performance data below overlaps due to the length of the line needed to show the complete output.

O|SS will gather information about the vector instructions that were executed in the application run, provided that a sample was taken at the address that corresponds to a vector instruction. There are three options that will enable this feature:

- `--vinstr128` Find vector instructions with operand sizes that are 128 bits or greater
- `--vinstr256` Find vector instructions with operand sizes that are 256 bits or greater
- `--vinstr512` Find vector instructions with operand sizes that are 512 bits or greater

For example: **osspsamp --vinstr512 "mpirun -np 256 ./smg2000 -n 5 5 5"**

This example was generated with `--vinstr128` and the view shows the time spent, percentage of time attributed to the instruction, name of the library or executable, vector instruction opcode and operands, and the maximum operand size for this instruction. The maximum operand size is the physical machine operand size, not the actual vector size at runtime. That is an extension that is not available at this time.

```
openss>>expview -vvectorinstr -f test_HPCCG

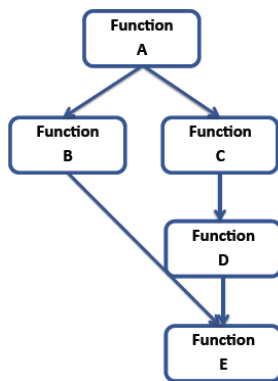
Exclusive CPU   % of CPU  Vector Instr Location (Line Number/Addr : OpCode : Max Operand Size (bits))
time in        Time
seconds.
2394.11000000  21.84626194  0x20007000 (HPC_sparsemv : test_HPCCG) : HPC_sparsemv.cpp(74) :
vmovsd %xmm0,0x0(%rsi,%rdi,8),%xmm7 : 128
2374.39000000  21.66631687  0x2000703a (HPC_sparsemv : test_HPCCG) : HPC_sparsemv.cpp(74) :
vmovsd %xmm0,0x0(%rsi,%rdi,8),%xmm11 : 128
2333.83000000  21.29620674  0x2000701d (HPC_sparsemv : test_HPCCG) : HPC_sparsemv.cpp(74) :
vmovsd %xmm0,0x0(%rsi,%rdi,8),%xmm9 : 128
450.93000000   4.11473779  0x20006ffa (HPC_sparsemv : test_HPCCG) : HPC_sparsemv.cpp(74) :
vfmadd231pd %xmm6,0x0(%rdx,%rbx,8),%xmm4 : 128
427.65000000   3.90230771  0x20007b8e (waxpby : test_HPCCG) : waxpby.cpp(57) : movsd %xmm0,0x0(%rax,%rbx,8) : 128
414.81000000   3.78514267  0x20007033 (HPC_sparsemv : test_HPCCG) : HPC_sparsemv.cpp(74) :
vfmadd231pd %xmm10,0x20(%rdx,%rbx,8),%xmm2 : 128
387.50000000   3.53593883  0x20006fe4 (HPC_sparsemv : test_HPCCG) : HPC_sparsemv.cpp(74) :
vmovsd %xmm0,0x0(%rsi,%rdi,8),%xmm5 : 128
382.54000000   3.49067881  0x20007016 (HPC_sparsemv : test_HPCCG) : HPC_sparsemv.cpp(74) :
vfmadd231pd %xmm8,0x10(%rdx,%rbx,8),%xmm3 : 128
330.18000000   3.01289363  0x20007b88 (waxpby : test_HPCCG) : waxpby.cpp(57) :
vfmadd213sd %xmm2,0x0(%rcx,%rbx,8),%xmm0 : 128
167.85000000   1.53163182  0x2000707d (HPC_sparsemv : test_HPCCG) : HPC_sparsemv.cpp(74) :
vmovsd %xmm0,0x0(%rsi,%rbx,8),%xmm2 : 128
163.58000000   1.49266806  0x2000704b (HPC_sparsemv : test_HPCCG) : HPC_sparsemv.cpp(74) :
vfmadd231pd %xmm12,0x30(%rdx,%rbx,8),%xmm1 : 128
133.74000000   1.22037796  0x20007067 (HPC_sparsemv : test_HPCCG) : HPC_sparsemv.cpp(64) :
vunpckhpd %xmm2,%xmm2,%xmm4 : 128
```

```

98.77000000 0.90127659 0x20007082 (HPC_sparsemv : test_HPCCG) : HPC_sparsemv.cpp(74) :
vmulsd %xmm2,0x0(%rdx,%rbp,8),%xmm3 : 128
95.85000000 0.87463158 0x20007063 (HPC_sparsemv : test_HPCCG) : HPC_sparsemv.cpp(64) :
vaddpd %xmm3,%xmm1,%xmm2 : 128
59.83000000 0.54594895 0x20007ba2 (waxpby : test_HPCCG) : waxpby.cpp(57) : movsd %xmm1,0x8(%rax,%rbx,8) : 128
55.69000000 0.50817144 0x20006f42 (HPC_sparsemv : test_HPCCG) : HPC_sparsemv.cpp(64) :
vxorpd %xmm1,%xmm1,%xmm1 : 128
43.98000000 0.40131765 0x2000700b (HPC_sparsemv : test_HPCCG) : HPC_sparsemv.cpp(74) : vmovhpd
0x0(%rsi,%rdi,8),%xmm7,%xmm8 : 128
42.12000000 0.38434514 0x20007045 (HPC_sparsemv : test_HPCCG) : HPC_sparsemv.cpp(74) : vmovhpd
0x0(%rsi,%rdi,8),%xmm11,%xmm12 : 128
40.88000000 0.37303014 0x2000706b (HPC_sparsemv : test_HPCCG) : HPC_sparsemv.cpp(64) :
vaddsd %xmm2,%xmm4,%xmm1 : 128
37.10000000 0.33853763 0x20006fef (HPC_sparsemv : test_HPCCG) : HPC_sparsemv.cpp(74) : vmovhpd
0x0(%rsi,%rdi,8),%xmm5,%xmm6 : 128
35.36000000 0.32266012 0x200089e6 (ddot : test_HPCCG) : ddot.cpp(56) :
vfmadd231pd %ymm6,0x40(%rax,%rdx,8),%ymm1 : 256
32.86000000 0.29984761 0x20007b94 (waxpby : test_HPCCG) : waxpby.cpp(57) :
vmovsd %xmm0,0x8(%rdx,%rbx,8),%xmm1 : 128
32.50000000 0.29656261 0x200089ed (ddot : test_HPCCG) : ddot.cpp(56) :
vfmadd231pd %ymm7,0x60(%rax,%rdx,8),%ymm0 : 256
32.10000000 0.29291261 0x200089d9 (ddot : test_HPCCG) : ddot.cpp(56) :
vfmadd231pd %ymm4,0x0(%rax,%rdx,8),%ymm3 : 256
31.34000000 0.28597761 0x200089df (ddot : test_HPCCG) : ddot.cpp(56) :
vfmadd231pd %ymm5,0x20(%rax,%rdx,8),%ymm2 : 256
30.28000000 0.27630510 0x20007b82 (waxpby : test_HPCCG) : waxpby.cpp(57) :
...
...
8.54000000 0.07792753 0x20006fd2 (HPC_sparsemv : test_HPCCG) : HPC_sparsemv.cpp(64) :
vmovdqa %xmm0,%xmm0,%xmm2 : 128
8.53000000 0.07783628 0x2000870e (ddot : test_HPCCG) : ddot.cpp(51) : vmovupd %ymm0,0x40(%rax,%rdi,8),%ymm6 :
256
8.41000000 0.07674128 0x2000871c (ddot : test_HPCCG) : ddot.cpp(51) : vfmadd231pd %ymm4,%ymm4,%ymm3 : 256
7.92000000 0.07227003 0x20008707 (ddot : test_HPCCG) : ddot.cpp(51) : vmovupd %ymm0,0x20(%rax,%rdi,8),%ymm5 :
256
7.72000000 0.07044503 0x20007b08 (waxpby : test_HPCCG) : waxpby.cpp(57) :
vfmadd213pd %ymm0,0x0(%rcx,%rcx,8),%ymm1 : 256
3.52000000 0.03212001 0x2000606c (generate_matrix : test_HPCCG) : generate_matrix.cpp(121) :
movsd %xmm0,(%edx) : 128
2.80000000 0.02555001 0x20008726 (ddot : test_HPCCG) : ddot.cpp(51) : vfmadd231pd %ymm6,%ymm6,%ymm1 : 256
2.73000000 0.02491126 0x20007b0e (waxpby : test_HPCCG) : waxpby.cpp(57) : movupd %ymm1,0x0(%rdx,%rcx,8) : 256
2.63000000 0.02399876 0x2000872b (ddot : test_HPCCG) : ddot.cpp(51) : vfmadd231pd %ymm7,%ymm7,%ymm0 : 256
2.30000000 0.02098751 0x20008721 (ddot : test_HPCCG) : ddot.cpp(51) : vfmadd231pd %ymm5,%ymm5,%ymm2 : 256
1.40000000 0.01277500 0x20007b9b (waxpby : test_HPCCG) : waxpby.cpp(57) :
vfmadd213sd %xmm2,0x8(%rcx,%rbx,8),%xmm1 : 128
...
...
0.01000000 0.00009125 0x20006123 (generate_matrix : test_HPCCG) : generate_matrix.cpp(136) :
vcvttsi2sd %xmm3,%xmm1 : 128
0.01000000 0.00009125 0x20008a66 (ddot : test_HPCCG) : ddot.cpp(56) : vmulpd %ymm1,0x0(%rax,%rdx,8),%ymm2 :
256
openss>>

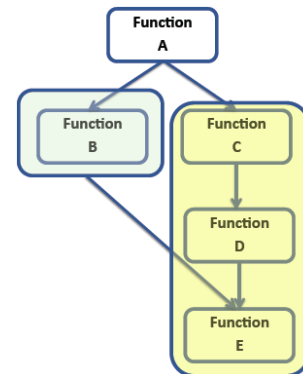
```

4.1 Call Path Profiling (usertime) Experiment



The call path profiling (usertime) experiment can add information that is missing from the flat profiles. It can distinguish routines called from multiple callers and understand the call invocation history, providing context for the performance data. It also gathers stack traces for each performance sample and only aggregates samples with equal stack traces. This simplifies the user's view by showing the caller/callee relationship. It also can highlight the hot call paths, the routes through the application that take the most time.

The call path profiling experiment also provides inclusive and exclusive time. Exclusive time is spent inside a function only, for example, in the graphic shown at the right, function B (blue box). Inclusive time is spent inside a function and its children, for example, the full chain of function C, D and E (yellow box).



The call path profiling experiment is similar to the program counter sampling experiment in that it collects program counter information. The difference is the call path profiling experiment collects call stack information at every sample.

There are, of course, tradeoffs: The user gets additional context information from the call stacks but incurs higher overhead with a necessarily lower sampling rate.

4.1.1 Call Path Profiling (usertime) experiment performance data gathering

We can use the OJSS convenience script to run the call path profiling experiment on our test program, SMG2000:

```
> ossusertime "mpirun -np 256 smg2000 -n 50 50 50"
```

Again, we recommend that users compile their code with the `-g` option to see the statements in the sampling. Sampling frequency also is an optional parameter in the usertime experiment, with settings of high (70 samples per second), low (18 samples per second) and default (35 samples per second). Note that these sample rates are lower than the pcsamp experiment because more data are collected. To run the same experiment with the low sampling rate, simply issue the command:

```
> ossusertime "mpirun -np 256 smg2000 -n 50 50 50" low
```

Here is an example run of a usertime experiment with full output:

```

ossusertime "mpirun -np 4 ./smg2000 -n 65 65 65"
[openss]: usertime experiment using the default sampling rate: "35".
Creating topology file for frontend host localhost
Generated topology file: ./cbtfAutoTopology
Running usertime collector.
Program: mpirun -np 4 ./smg2000 -n 65 65 65
Number of mrnet backends: 4
Topology file used: ./cbtfAutoTopology
executing mpi program: mpirun -np 4 cbtfrun --mpi --mrnet -c usertime ./smg2000 -n 65 65 65
Running with these driver parameters:
  (nx, ny, nz) = (65, 65, 65)
  (Px, Py, Pz) = (4, 1, 1)
  (bx, by, bz) = (1, 1, 1)
  (cx, cy, cz) = (1.000000, 1.000000, 1.000000)
  (n_pre, n_post) = (1, 1)
  dim = 3
  solver ID = 0
=====
Struct Interface:
=====
Struct Interface:
  wall clock time = 0.023957 seconds
  cpu clock time = 0.030000 seconds
=====
Setup phase times:
=====
SMG Setup:
  wall clock time = 0.594738 seconds
  cpu clock time = 0.590000 seconds
=====
Solve phase times:
=====
SMG Solve:
  wall clock time = 4.306247 seconds
  cpu clock time = 4.280000 seconds

Iterations = 7
Final Relative Residual Norm = 1.760588e-07

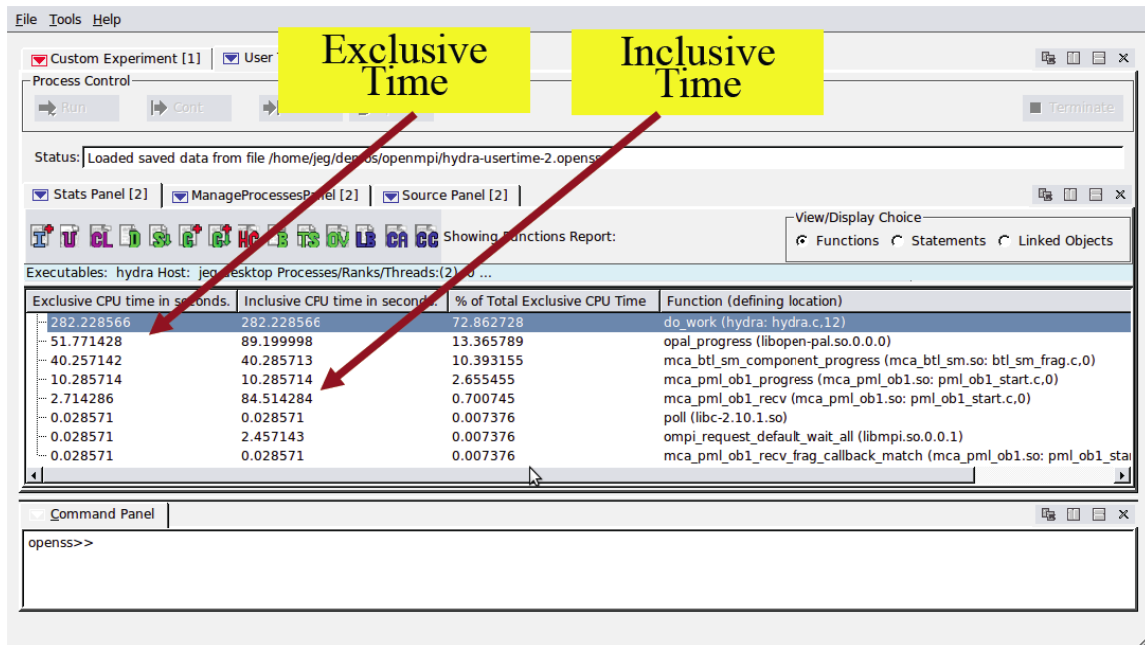
All Threads are finished.
default view for ./smg2000-usertime-14.openss
[openss]: The restored experiment identifier is: -x 1
Performance data spans 5.271756 seconds from 2016/11/11 08:06:12 to 2016/11/11 08:06:17

Exclusive Inclusive % of Function (defining location)
CPU time CPU time Total
in in Exclusive
seconds. seconds. CPU Time
9.000000 9.800000 45.718433 hypre_SMGResidual (smg2000: smg_residual.c,152)
4.171428 7.057143 21.190131 hypre_CyclicReduction (smg2000: cyclic_reduction.c,757)
0.542857 0.571429 2.757620 hypre_SemiInterp (smg2000: semi_interp.c,126)
0.514286 1.542857 2.612482 mca_btl_vader_check_fboxes (libmpi.so.12.0.2: btl_vader_fbox.h,184)
0.514286 0.942857 2.612482 pack_predefined_data (libopen-pal.so.13.0.2: opal_datatype_pack.h,35)
0.485714 0.485714 2.467344 __memcpy_ssse3_back (libc-2.17.so)
0.457143 0.514286 2.322206 unpack_predefined_data (libopen-pal.so.13.0.2: opal_datatype_unpack.h,34)
0.314286 2.314286 1.596517 opal_progress (libopen-pal.so.13.0.2: opal_progress.c,151)
0.285714 0.314286 1.451379 hypre_SemiRestrict (smg2000: semi_restrict.c,125)
0.257143 0.257143 1.306241 hypre_StructAxy (smg2000: struct_axpy.c,25)
0.228571 0.228571 1.161103 hypre_SMGAxy (smg2000: smg_axpy.c,27)
0.171429 0.171429 0.870827 hypre_SMGSetStructVectorConstantValues (smg2000: smg.c,379)

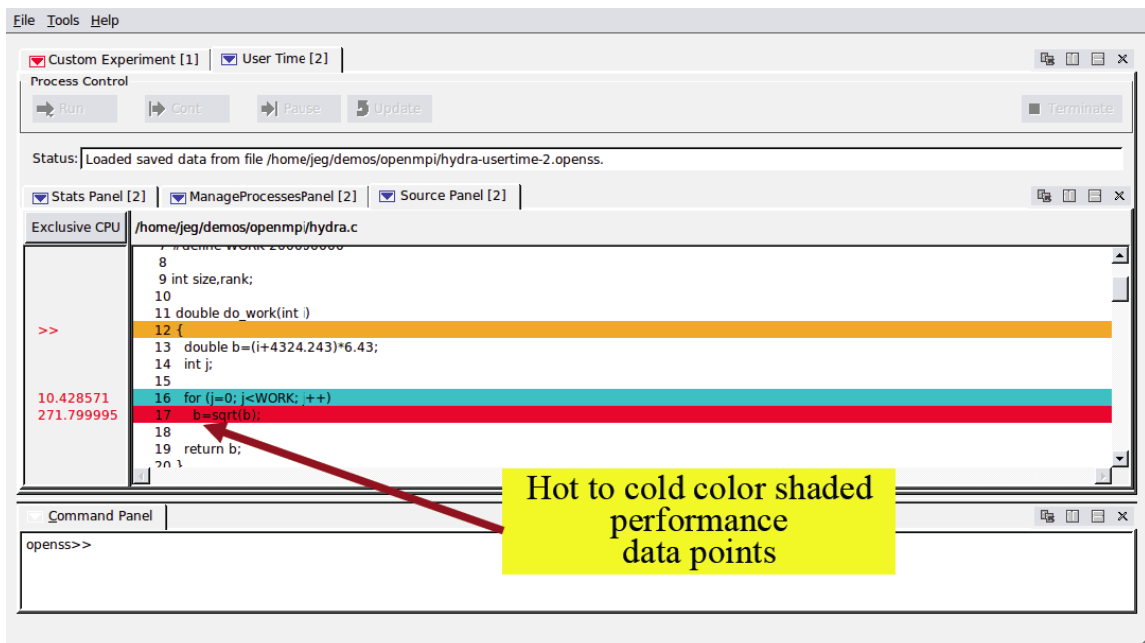
```

4.1.2 Viewing Call Path Profiling (usertime) experiment performance data via GUI

Users can view results of this experiment in the OJSS GUI. The view is similar to that in pcsamp but in this case the inclusive CPU time also is shown.

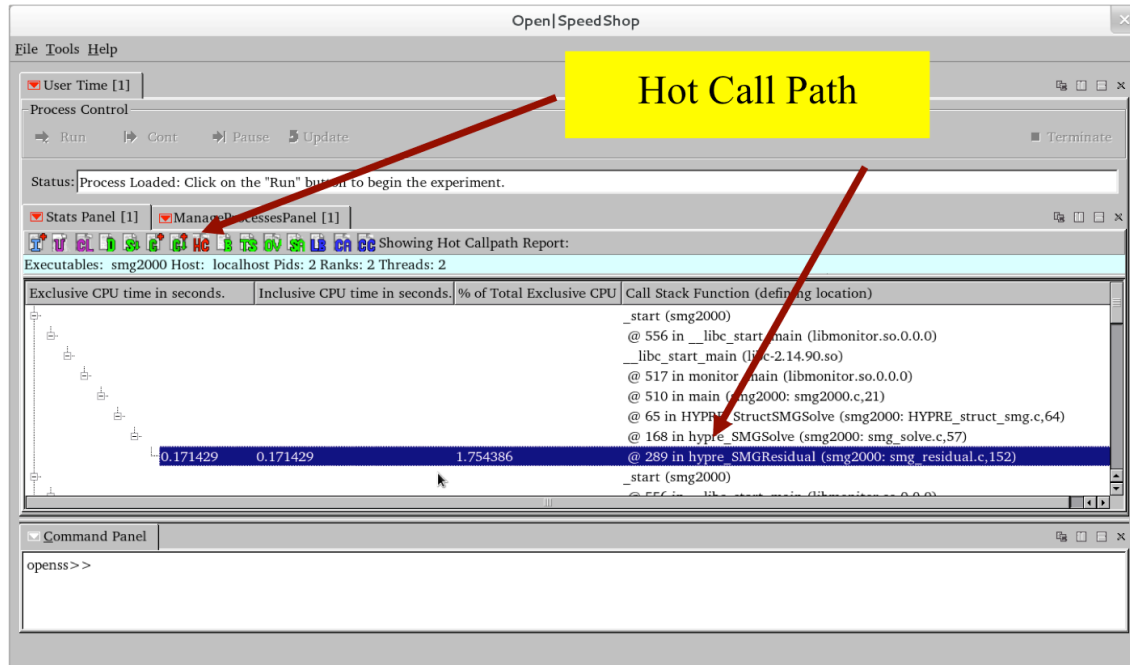


Below, the Exclusive CPU time is shown on highlighted lines that indicate relatively high CPU times.

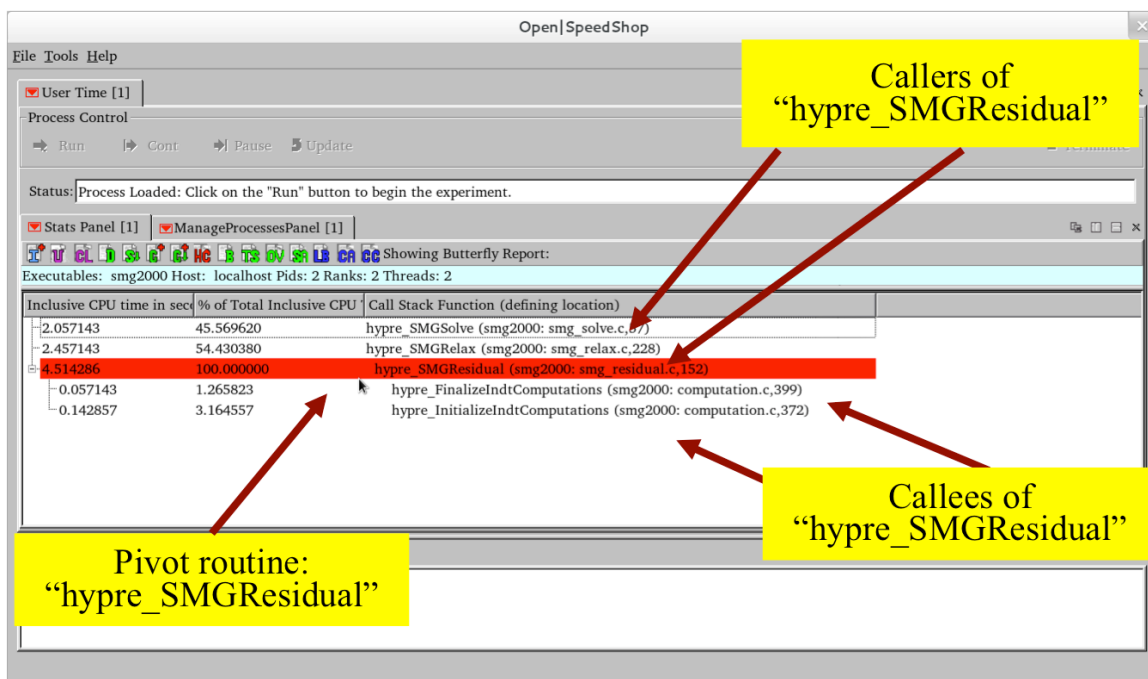


While performance tools will highlight potential bottlenecks and hot areas, it is still up to the user to interpret most data in the correct context and to note code areas they may want to probe further. If the inclusive and exclusive times are similar, it

means the child executions are insignificant (with respect to CPU time) and profiling below this layer may not be useful. If the inclusive time is significantly greater than the exclusive time, then users should focus their attention on execution times for the children.



The stack trace views in O|SS are similar to the well-known Unix profiling tool gprof.



4.1.3 Viewing Call Path Profiling (usertime) experiment performance data via CLI

This table describes information included in the usertime experiment default view.

Column Name	Column Definition
Exclusive CPU Time	Aggregated total exclusive time spent in the application function corresponding to this row of data.
% of CPU Time	Percentage of exclusive time spent in the function corresponding to this row of data relative to the total application exclusive time for all the application functions.
Inclusive CPU Time	Aggregated total inclusive time spent in the application function corresponding to this row of data.

To load a database file into the CLI, use this form of the openss client:

```
$ openss -cli -f./smg2000-usertime-14.openss
```

This is a default CLI view for the usertime experiment, restricted to the top 10 time-consuming functions. Using the experiment name and the number of items to be displayed limits the output to those items.

```
openss>>expview usertime10
```

Exclusive CPU time in seconds.	Inclusive CPU time in seconds.	% of Total Exclusive CPU Time	Function (defining location)
9.000000	9.800000	45.718433	hypr_SMGResidual (smg2000: smg_residual.c,152)
4.171428	7.057143	21.190131	hypr_CyclicReduction (smg2000: cyclic_reduction.c,757)
0.542857	0.571429	2.757620	hypr_SemiInterp (smg2000: semi_interp.c,126)
0.514286	1.542857	2.612482	mca_btl_vader_check_fboxes (libmpi.so.12.0.2: btl_vader_fbox.h,184)
0.514286	0.942857	2.612482	pack_predefined_data (libopen-pal.so.13.0.2: opal_datatype_pack.h,35)
0.485714	0.485714	2.467344	__memcpy_ssse3_back (libc-2.17.so)
0.457143	0.514286	2.322206	unpack_predefined_data (libopen-pal.so.13.0.2: opal_datatype_unpack.h,34)
0.314286	2.314286	1.596517	opal_progress (libopen-pal.so.13.0.2: opal_progress.c,151)
0.285714	0.314286	1.451379	hypr_SemiRestrict (smg2000: semi_restrict.c,125)
0.257143	0.257143	1.306241	hypr_StructAxy (smg2000: struct_axpy.c,25)

The display below shows the top 10 time-consuming statements in the program:

```
openss>>expview -v statements usertime10
```

Exclusive CPU time in seconds.	Inclusive CPU time in seconds.	% of Total CPU Time	Statement Location (Line Number)
7.085714	7.085714	41.471572	smg_residual.c(289)
1.371429	1.371429	8.026756	cyclic_reduction.c(1130)
1.314286	1.314286	7.692308	smg_residual.c(238)
0.828571	0.828571	4.849498	cyclic_reduction.c(910)
0.485714	0.485714	2.842809	cyclic_reduction.c(999)
0.285714	0.285714	1.672241	smg_residual.c(287)
0.285714	0.285714	1.672241	cyclic_reduction.c(853)
0.285714	0.285714	1.672241	cyclic_reduction.c(1000)
0.257143	0.257143	1.505017	semi_interp.c(294)
0.228571	0.228571	1.337793	opal_datatype_unpack.h(59)

The top 10 time-consuming loops in the application are shown below. The line number (for example: 204 in first entry) is for the line in which the loop begins. The static analysis does not provide the loop's ending-line number, but it is the line that corresponds to the end of the logical loop construct.

```
openss>>expview -v loops usertime10
```

Exclusive CPU time in seconds.	Inclusive CPU time in seconds.	% of Total CPU Time	Loop Start Location (Line Number)
8.971428	9.771428	37.649880	smg_residual.c(204)
1.914286	3.257143	8.033573	cyclic_reduction.c(1022)
1.885714	3.400000	7.913669	cyclic_reduction.c(882)
0.857143	0.885714	3.597122	cyclic_reduction.c(889)
0.485714	1.514286	2.038369	btl_vader_fbox.h(188)
0.400000	0.828571	1.678657	opal_datatype_pack.h(59)
0.371429	0.428571	1.558753	opal_datatype_unpack.h(58)
0.371429	0.400000	1.558753	semi_interp.c(238)
0.371429	0.400000	1.558753	cyclic_reduction.c(835)
0.371429	0.400000	1.558753	semi_interp.c(258)

```
openss>>
```

This shows time spent in the application libraries:

```
openss>>expview -v linkedobjects
```

Exclusive CPU time in seconds.	Inclusive CPU time in seconds.	% of Total CPU Time	LinkableObject
15.514285	19.742857	78.581766	smg2000
1.800000	3.400000	9.117221	libopen-pal.so.13.0.2
1.400000	3.885714	7.091172	libmpi.so.12.0.2
1.028571	19.742857	5.209841	libc-2.17.so

This shows the application's top three time-consuming call paths:

```
openss>>expview -v fullstack usertime3
```

Exclusive CPU time in seconds.	Inclusive CPU time in seconds.	% of Total CPU Time	Call Stack Function (defining location)
			_start (smg2000)
			> @ 556 in __libc_start_main (libmonitor.so.0.0.0: main.c,541)
			>> __libc_start_main (libc-2.17.so)
			>>> @ 517 in monitor_main (libmonitor.so.0.0.0: main.c,492)
			>>>> @ 510 in main (smg2000: smg2000.c,21)
			>>>>> @ 65 in HYPRE_StructSMGSolve (smg2000: HYPRE_struct_smg.c,64)
			>>>>>> @ 224 in hypre_SMGSolve (smg2000: smg_solve.c,57)
0.800000	0.800000	4.052098	>>>>>> @ 289 in hypre_SMGResidual (smg2000: smg_residual.c,152)
			_start (smg2000)
			> @ 556 in __libc_start_main (libmonitor.so.0.0.0: main.c,541)
			>> __libc_start_main (libc-2.17.so)
			>>> @ 517 in monitor_main (libmonitor.so.0.0.0: main.c,492)
			>>>> @ 510 in main (smg2000: smg2000.c,21)
			>>>>> @ 65 in HYPRE_StructSMGSolve (smg2000: HYPRE_struct_smg.c,64)
			>>>>>> @ 168 in hypre_SMGSolve (smg2000: smg_solve.c,57)
0.400000	0.400000	2.026049	>>>>>> @ 289 in hypre_SMGResidual (smg2000: smg_residual.c,152)
			_start (smg2000)
			> @ 556 in __libc_start_main (libmonitor.so.0.0.0: main.c,541)
			>> __libc_start_main (libc-2.17.so)
			>>> @ 517 in monitor_main (libmonitor.so.0.0.0: main.c,492)
			>>>> @ 510 in main (smg2000: smg2000.c,21)
			>>>>> @ 65 in HYPRE_StructSMGSolve (smg2000: HYPRE_struct_smg.c,64)
			>>>>>> @ 164 in hypre_SMGSolve (smg2000: smg_solve.c,57)
			>>>>>>> @ 325 in hypre_SMGRelax (smg2000: smg_relax.c,228)
			>>>>>>>> @ 224 in hypre_SMGSolve (smg2000: smg_solve.c,57)
0.400000	0.400000	2.026049	>>>>>>>> @ 289 in hypre_SMGResidual (smg2000: smg_residual.c,152)

This butterfly view shows the functions calling hypre_SMGSolve along with the functions hypre_SMGSolve calls. Hypre_SMGSolve is the pivot point in this view.

```
openss>>expview -vbutterfly -f hypre_SMGSolve
```

Inclusive CPU time in seconds.	% of Total Inclusive CPU Time	Call Stack Function (defining location)
17.200000	94.654088	<HYPRE_StructSMGSolve (smg2000: HYPRE_struct_smg.c,64)
0.971429	5.345912	<hypre_SMGRelax (smg2000: smg_relax.c,228)
18.171428	100.000000	hypre_SMGSolve (smg2000: smg_solve.c,57)
16.285714	89.622642	>hypre_SMGRelax (smg2000: smg_relax.c,228)
1.428571	7.861635	>hypre_SMGResidual (smg2000: smg_residual.c,152)
0.171429	0.943396	>hypre_SemiInterp (smg2000: semi_interp.c,126)
0.114286	0.628931	>hypre_SemiRestrict (smg2000: semi_restrict.c,125)

```
0.114286 0.628931 >hydre_StructAxy (smg2000: struct_axpy.c,25)
0.057143 0.314465 >hydre_StructInnerProd (smg2000: struct_innerprod.c,32)
```

This displays only the percentage performance information because the user gave the metric “percent” to the expview command:

```
openss>>expview -m percent usertime9
```

```
% of Function (defining location)
Total
Exclusive
Time
45.718433 hydre_SMGResidual (smg2000: smg_residual.c,152)
21.190131 hydre_CyclicReduction (smg2000: cyclic_reduction.c,757)
2.757620 hydre_SemiInterp (smg2000: semi_interp.c,126)
2.612482 mca_btl_vader_check_fboxes (libmpi.so.12.0.2: btl_vader_fbox.h,184)
2.612482 pack_predefined_data (libopen-pal.so.13.0.2: opal_datatype_pack.h,35)
2.467344 __memcpy_ssse3_back (libc-2.17.so)
2.322206 unpack_predefined_data (libopen-pal.so.13.0.2: opal_datatype_unpack.h,34)
1.596517 opal_progress (libopen-pal.so.13.0.2: opal_progress.c,151)
1.451379 hydre_SemiRestrict (smg2000: semi_restrict.c,125)
```

4.1.4 Call Path Profiling (usertime) experiment function inline display

As of O|SS version 2.4.0, O|SS now displays the actual path to the point where the inline functions were inlined, thus giving a more complete picture of the call paths for C++.

The example call-stack below is from lulesh2.0.3 and was run on snow at SNL. This call-stack, which shows the number of calls metric, illustrates the relationship between the function doing the inlining and the inlined functions. The call-stack below chains together the inlined functions as they occurred in the execution of the program based on sampling information collected by O|SS.

This call-stack is one of the many that were collected and displayed by the usertime experiment:

```
_start (lulesh2.0)
> @ 556 in __libc_start_main (libmonitor.so.0.0.0)
>> @ 274 in __libc_start_main (libc-2.17.so)
>>> @ 517 in monitor_main (libmonitor.so.0.0.0)
>>>> @ 1609 inline CalcKinematicsForElems (/home/fred/src/lulesh2.0.3/lulesh.cc)
>>>> @ 2458 inline CalcLagrangeElements (/home/fred/src/lulesh2.0.3/lulesh.cc)
>>>> @ 2656 inline LagrangeElements (/home/fred/src/lulesh2.0.3/lulesh.cc)
>>>> @ 2774 inline LagrangeLeapFrog (/home/fred/src/lulesh2.0.3/lulesh.cc)
>>>> @ NN inlined main (lulesh2.0: lulesh.cc,2690)
>>>>> @ 2374 in __kmp_join_call (libomp.so)
```

```
>>>>> @ 7165 in __kmp_internal_join (libomp.so)
>>>>> @ 334 in __kmp_join_barrier(int) (libomp.so)
169 >>>>>> @ 167 in OMPT_THREAD_WAIT_BARRIER (collector.c,167)
```

The chain of events that are illustrated above:

- At line 2774 of lulesh.cc, LagrangeLeapFrog was inlined.
- At line 2656 in LagrangeLeapFrog, LagrangeElements was inlined
- At line 2458 in LagrangeElements, CalcLagrangeElements was inlined
- At line 1609 in CalcLagrangeElements, CalcKinematicsForElems was inlined

Source excerpts from lulesh.cc from lulesh2.0.3 that display the inline points of interest:

```
1598 /*****/
1599
1600 static inline
1601 void CalcLagrangeElements(Domain& domain, Real_t* vnew)
1602 {
1603     Index_t numElem = domain.numElem() ;
1604     if (numElem > 0) {
1605         const Real_t deltatime = domain.deltatime() ;
1606
1607         domain.AllocateStrains(numElem);
1608
1609     CalcKinematicsForElems(domain, vnew, deltatime, numElem) ;
1610
1611     // element loop to do some stuff not included in the elemLib function.
1612     #pragma omp parallel for firstprivate(numElem)
1613     for ( Index_t k=0 ; k<numElem ; ++k )
1614     {
1615         // calc strain rate and apply as constraint (only done in FB element)
1616
1617     ...
1618
1619     ...
1620
1621     }
1622     domain.DeallocateStrains();
1623 }
1624 }
1625 }
```

```
2451 /*****/
```

```

2452
2453 static inline
2454 void LagrangeElements(Domain& domain, Index_t numElem)
2455 {
2456   Real_t *vnew = Allocate<Real_t>(numElem) ; /* new relative vol -- temp */
2457
2458   CalcLagrangeElements(domain, vnew) ;
2459
2460   /* Calculate Q. (Monotonic q option requires communication) */
2461   CalcQForElems(domain, vnew) ;
2462
2463   ApplyMaterialPropertiesForElems(domain, vnew) ;
2464
2465   UpdateVolumesForElems(domain, vnew,
2466                         domain.v_cut(), numElem) ;
2467
2468   Release(&vnew);
2469 }
2470

```

```

2638
2639 static inline
2640 void LagrangeLeapFrog(Domain& domain)
2641 {
2642   #ifdef SEDOV_SYNC_POS_VEL_LATE
2643     Domain_member fieldData[6] ;
2644   #endif
2645
2646   /* calculate nodal forces, accelerations, velocities, positions, with
2647    * applied boundary conditions and slide surface considerations */
2648   LagrangeNodal(domain);
2649
2650
2651   #ifdef SEDOV_SYNC_POS_VEL_LATE
2652   #endif
2653
2654   /* calculate element quantities (i.e. velocity gradient & q), and update
2655    * material states */
2656   LagrangeElements(domain, domain.numElem());
2657
2658   #if USE_MPI
2659   #ifdef SEDOV_SYNC_POS_VEL_LATE
2660     CommRecv(domain, MSG_SYNC_POS_VEL, 6,
2661              domain.sizeX() + 1, domain.sizeY() + 1, domain.sizeZ() + 1,

```



```

2662         false, false) ;
2663
2664     fieldData[0] = &Domain::x ;
2665     fieldData[1] = &Domain::y ;
2666     fieldData[2] = &Domain::z ;
2667     fieldData[3] = &Domain::xd ;
2668     fieldData[4] = &Domain::yd ;
2669     fieldData[5] = &Domain::zd ;
2670
2671     CommSend(domain, MSG_SYNC_POS_VEL, 6, fieldData,
2672             domain.sizeX() + 1, domain.sizeY() + 1, domain.sizeZ() + 1,
2673             false, false) ;
2674 #endif
2675 #endif
2676
2677     CalcTimeConstraintsForElems(domain);
2678
2679 #if USE_MPI
2680 #ifdef SEDOV_SYNC_POS_VEL_LATE
2681     CommSyncPosVel(domain) ;
2682 #endif
2683 #endif
2684 }
2685
2686

```

```

2687 /*****/
2688
2689 int main(int argc, char *argv[])
2690 {
2691     Domain *locDom ;
2692     Int_t numRanks ;
2693     Int_t myRank ;
2694     struct cmdLineOpts opts;
2695
2696 #if USE_MPI
2697     Domain_member fieldData ;
2698
...
...

```

```

2770 //    std::cout << "region" << i + 1 << "size" << locDom->regElemSize(i)
<<std::endl;
2771 while((locDom->time() < locDom->stoptime()) && (locDom->cycle() <
opts.its)) {
2772
2773     TimeIncrement(*locDom);
2774     LagrangeLeapFrog(*locDom);
2775
2776     if ((opts.showProg != 0) && (opts.quiet == 0) && (myRank == 0)) {
2777         printf("cycle = %d, time = %e, dt=%e\n",
2778             locDom->cycle(), double(locDom->time()), double(locDom-
>deltatime()) );
2779     }
2780 }

```

4.1.4.1 Call Path Profiling (usertime) experiment function inline display: Specific Kokkos Example:

BEFORE changes to better support C++ inlining – top time taking callstack from kokkos-mxm.host:

openss>>expview -vfullstack -mcalls usertime1

Number of Call Stack Function (defining location)

Exclusive

Counts

```

_start (kokkos-mxm.host)
> @ 556 in __libc_start_main (libmonitor.so.0.0.0: main.c,541)
>> __libc_start_main (libc-2.26.so)
>>> @ 517 in monitor_main (libmonitor.so.0.0.0: main.c,492)
>>>> @ 131 in main (kokkos-mxm.host: kokkos-mxm.cpp,3)
>>>>> @ 1168 in __kmp_api_GOMP_parallel (libomp.so: kmp_gsupport.cpp,1136)
158 >>>>>> @ 91 in Kokkos::Impl::ParallelFor<main::{lambda(int const&)#2},
Kokkos::RangePolicy<Kokkos::OpenMP>, Kokkos::RangePolicy>::execute() const [clone ._omp_fn.2]
(kokkos-mxm.host: Kokkos_OpenMP_Parallel.hpp,131)

```

AFTER changes to better support C++ inlining – top time taking callstack from kokkos-mxm.host:

openss>>expview -v fullstack -mcalls usertime1

Number of Call Stack Function (defining location)

Exclusive

Counts

```

_start (kokkos-mxm.host)
> @ 556 in __libc_start_main (libmonitor.so.0.0.0)
>> __libc_start_main (libc-2.26.so)

```

```

>>> @ 517 in monitor_main (libmonitor.so.0.0.0)
>>>> @ 3 in main (kokkos-mxm.host: kokkos-mxm.cpp,3)
>>>> @ 244 in parallel_for<main(int, char**)::<lambda(int const&)>> > (kokkos-mxm.host:
Kokkos_Parallel.hpp)
>>>> @ 224 in execute (kokkos-mxm.host: Kokkos_Parallel.hpp)
>>>> @ 20 in parallel_for<int, main(int, char**)::<lambda(int const&)>> > (kokkos-mxm.host:
kokkos-mxm.cpp)
>>>>> @ 1168 in __kmp_api_GOMP_parallel (libomp.so)
138 >>>>>> @ 91 in Kokkos::Impl::ParallelFor<main::{lambda(int const&)#2},
Kokkos::RangePolicy<Kokkos::OpenMP>, Kokkos::RangePolicy>::execute() const [clone ._omp_fn.2] (kokkos-
mxm.host: Kokkos_OpenMP_Parallel.hpp,131)

```

4.1.4.2 Call Path Profiling (usertime) experiment function inline display: Specific Raja Example:

BEFORE changes to better support C++ inlining – top time taking callstack from rajaperf.exe:

openss>>expview -v fullstack -mcalls usertime1

Number of Call Stack Function (defining location)

Exclusive

Counts

```

_start (raja-perf.exe)
> @ 556 in __libc_start_main (libmonitor.so.0.0.0: main.c,541)
>>_libc_start_main (libc-2.26.so)
>>> @ 517 in monitor_main (libmonitor.so.0.0.0: main.c,492)
>>>> @ 34 in main (raja-perf.exe: RAJAPerfSuiteDriver.cpp,22)
>>>>> @ 372 in rajaperf::Executor::runSuite() (raja-perf.exe: Executor.cpp,348)
>>>>>> @ 72 in rajaperf::KernelBase::execute(rajaperf::VariantID) (raja-perf.exe:
KernelBase.cpp,64)
73 >>>>>>> @ 89 in rajaperf::stream::MUL::runKernel(rajaperf::VariantID) (raja-perf.exe:
MUL.cpp,56)

```

AFTER changes to better support C++ inlining – top time taking callstack from rajaperf.exe:

Number of Call Stack Function (defining location)

Exclusive

Counts

```

_start (raja-perf.exe)
> @ 556 in __libc_start_main (libmonitor.so.0.0.0)
>>_libc_start_main (libc-2.26.so)
>>> @ 517 in monitor_main (libmonitor.so.0.0.0)
>>>> @ 34 in main (raja-perf.exe: RAJAPerfSuiteDriver.cpp,22)
>>>>> @ 372 in rajaperf::Executor::runSuite() (raja-perf.exe: Executor.cpp,348)
>>>>>> @ 72 in rajaperf::KernelBase::execute(rajaperf::VariantID) (raja-perf.exe:
KernelBase.cpp,64)

```

```

78 >>>>>> @ 56 in rajaperf::stream::MUL::runKernel(rajaperf::VariantID) (raja-perf.exe:
MUL.cpp,56)
>>>>>> @ 66 in operator() (raja-perf.exe:
/home/jeg/raja/RAJAPerf/tpl/RAJA/include/RAJA/policy/simd/forall.hpp)
>>>>>> @ 740 in forall<RAJA::policy::loop::loop_exec, RAJA::TypedRangeSegment<long int>,
rajaperf::apps::PRESSURE::runKernel(rajaperf::VariantID)::<lambda(int)> > (raja-perf.exe:
/home/jeg/raja/RAJAPerf/tpl/RAJA/include/RAJA/pattern/forall.hpp)
>>>>>> @ 399 in forall<RAJA::policy::loop::loop_exec, RAJA::TypedRangeSegment<long int>,
rajaperf::apps::PRESSURE::runKernel(rajaperf::VariantID)::<lambda(int)> > (raja-perf.exe:
/home/jeg/raja/RAJAPerf/tpl/RAJA/include/RAJA/pattern/forall.hpp)
>>>>>> @ 214 in forall_impl<RAJA::TypedRangeSegment<long int>,
rajaperf::apps::PRESSURE::runKernel(rajaperf::VariantID)::<lambda(int)>&> (raja-perf.exe:
/home/jeg/raja/RAJAPerf/tpl/RAJA/include/RAJA/pattern/forall.hpp)
>>>>>> @ 87 in forall<RAJA::policy::simd::simd_exec, RAJA::TypedRangeSegment<long int,
long int>, rajaperf::stream::MUL::runKernel(rajaperf::VariantID)::<lambda(rajaperf::Index_type)> > (raja-
perf.exe: /home/jeg/raja/RAJAPerf/src/stream/MUL.cpp)

```

5 How to Relate Data to Architectural Properties

Performance Application Programming Interface (PAPI) allows access to hardware counters through APIs and simple runtime tools. Find more about PAPI at <http://icl.cs.utk.edu/papi>.

OJSS implements three hardware counter experiments on top of PAPI. It provides access to PAPI and native counters like data cache misses, TLB misses and bus accesses.

There are a few basic models to follow in hardware counter experiments. The first is thresholding: The user selects a counter and the application runs until the counter reaches a fixed number of events. A PC sample is then taken at that location every time the counter increases by the preset fixed number. The ideal threshold (the fixed number at which to monitor) depends on the application. Another model is a timer-based sampling in which the counters are checked at given time intervals.

OJSS provides three hardware counter experiments: `hwc` for flat hardware counter profiles using a single hardware counter; `hwctime` for profiles with stack traces using a single hardware counter; and `hwcsamp` for PC sampling with multiple hardware counters. Both `osshwc` and `osshwctime` support non-derived PAPI presets: All non-derived events are reported by “`papi_avail -a`”. Users also can see the available events by running the experiments (`osshwc` or `osshwctime`) with no arguments. The experiments include all native events for that specific architecture. Some PAPI event names are in the sections below, but please see the PAPI documentation for the full list.

The threshold chosen depends on the application; users should balance overhead with accuracy. Remember: a higher threshold will record fewer samples; rare events need a smaller threshold or that information may be lost (never triggered and recorded). Use a larger threshold for frequent events to reduce the overhead of collecting the information. Selecting the right threshold can take experience or some trial and error.

HINT: Running the sampling-based hardware counter experiment, `osshwcsamp`, can help suggest a threshold value to try when running the threshold-based `osshwc` and `osshwctime` experiments. Since the ideal number of events (the threshold) depends on the application and the selected counter, the `hwcsamp` experiment can be used to get an overview of counter activity for events other than the default.

The default threshold is set to a high value to match the default event (`PAPI_TOT_CYC`). For all other events, the user should run `hwcsamp` first to understand how many times a particular event occurs (the count of the event) during the program’s life. To ascertain a reasonable threshold from the `hwcsamp` data, determine the average counts per thread of execution and then set the

hwc/hwctime threshold to some small fraction of that. For example, if there are 1333333333 PAPI_L1_DCM's over the program's life when running the hwcsamp experiment and there were 524 processes used during the application run, the following formula could find a reasonable threshold for the hwc and hwctime experiments when using the PAPI_L1_DCM event for the same application:

$$(\text{Average counts per thread}) / 1000 == \text{Threshold for hwc/hwctime}$$

In this case:

$$(1333333333/524)/1000 == 2544529/1000 == 2545$$

With this formula, a user could choose 2545 as the threshold value in hwc and hwctime for PAPI_L1_DCM and expect to get a reasonable data sample.

NOTE: The number of PAPI counters and their uses can be overwhelming. Ratios derived from a combination of hardware events sometimes can provide more useful information than raw metrics. Develop the ability to interpret metric ratios with a focus on understanding:

- Instructions per cycle or cycles per instruction.
- Floating point/vectorization efficiency.
- Cache behaviors; long latency instruction impact.
- Branch mispredictions.
- Memory and resource access patterns.
- Pipeline stalls.

5.1 Hardware Counter Experiment (hwc)

As an example, here's a run of the osshwc experiment on our test program, SMG2000. The convenience script for this experiment is:

```
> osshwc "mpirun -np 256 smg2000 -n 50 50 50" <counter> <threshold>
```

This is the same syntax as the osshwctime experiment. Note: If the output is empty, try lowering the <threshold> value; OJSS calculates it by default. Users can try lowering the threshold value if there have not been enough PAPI event occurrences to record. Also, see the hint in the osshwcsamp section above. Users can run osshwcsamp and use a formula to create a reasonable threshold. Any counter reported by "papi_avail -a" that is not derived is available for use. Users also can see the available counters by using the osshwc or osshwctime commands with no arguments. Native counters are listed in the PAPI documentation.

PAPI Name	Description	Threshold
PAPI_L1_DCM	L1 data cache misses	high
PAPI_L2_DCM	L2 data cache misses	high/medium
PAPI_L1_DCA	L1 data cache accesses	high
PAPI_FPU_IDL	Cycles in which FPUs are idle	high/medium
PAPI_STL_ICY	Cycles with no instruction issue	high/medium
PAPI_BR_MSP	Miss-predicted branches	medium/low
PAPI_FP_INS	Number of floating point instructions	high
PAPI_LD_INS	Number of load instructions	high
PAPI_VEC_INS	Number of vector/SIMD instructions	high/medium
PAPI_HW_INT	Number of hardware interrupts	low
PAPI_TLB_TL	Number of TLB misses	low

Note: Threshold indications are just for rough guidance and depend on the application. Also, remember that not all counters will exist on all platforms. Run `osshwc` with no arguments to see the available hardware counters.

The sections below show outputs from the `osshwc` experiment. Note that the default counter is the total cycles.

5.1.1 Hardware Counter Threshold (hwc) experiment performance data gathering

The hardware counter threshold experiment convenience script is “`osshwc`”. Here’s how to use this to gather counter values for one unique hardware counter:

```
osshwc “how you normally run your application” <papi event > < threshold value>
```

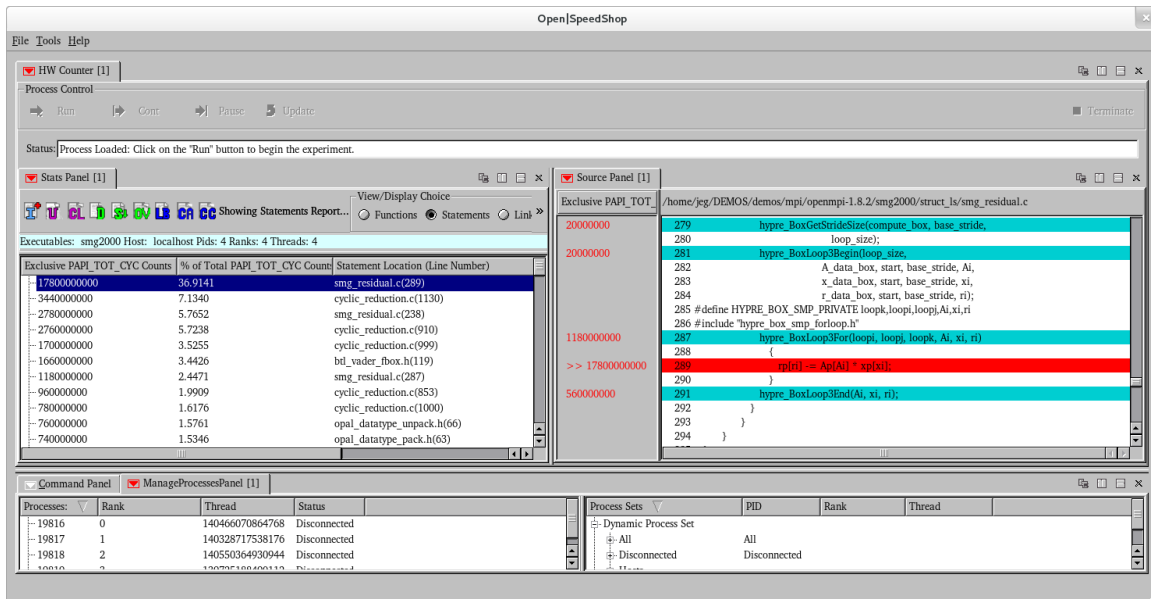
Here’s an example of how to gather data for the SMG2000 application on a Linux cluster platform using the `osshwc` convenience script. It collects performance data for the default counter, `PAPI_TOT_CYC` because there is no hardware counter value specified after the quoted application run command:

```
osshwc “mpirun -np 4 ./smg2000 -n 60 60 60”
```

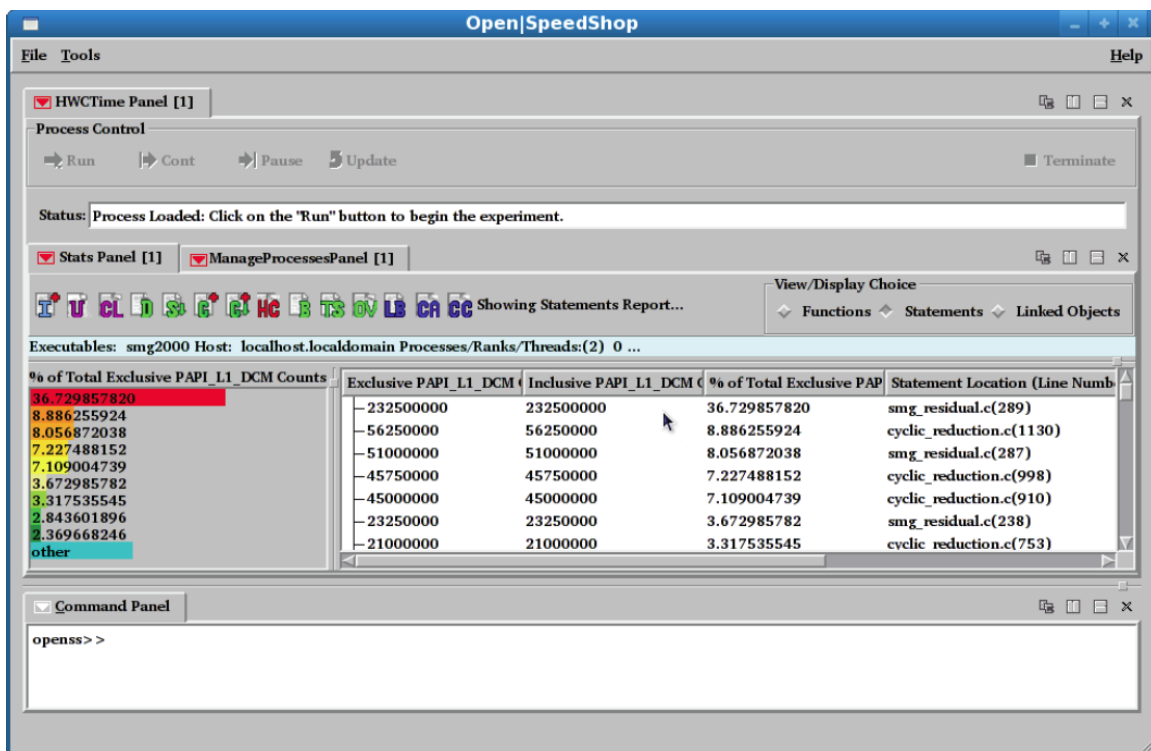
5.1.2 Viewing Hardware Counter Threshold (hwc) experiment performance data via GUI

To launch the GUI on any experiment, use “`openss -f <database name>`”.

The image below shows the default view for the hwc experiment run with the SMG2000 MPI application using PAPI_TOT_CYC as the hardware counter event. Double-clicking on a performance information line in the Stats Panel or on the bar chart will take the user to the source file and line it represents.



This displays output from the osshwctime experiment in which the counter is the L1 cache misses:



5.1.3 Viewing Hardware Counter Threshold (hwc) experiment performance data via CLI

To launch the CLI on any experiment, use “`openss -cli -f <database name>`”. This example shows three default CLI views of varying granularities: function, statement and library level.

```
openss -f smg2000-hwc-3.openss
[openss]: The restored experiment identifier is: -x 1

$ openss -cli -f smg2000-hwc-3.openss
openss>>[openss]: The restored experiment identifier is: -x 1
openss>>expview
```

Exclusive PAPI_TOT_CYC Counts	% of Total PAPI_TOT_CYC Counts	Function (defining location)
23080000000	43.8283	hypr_SMGResidual (smg2000: smg_residual.c,152)
12880000000	24.4588	hypr_CyclicReduction (smg2000: cyclic_reduction.c,757)
3540000000	6.7224	mca_btl_vader_check_fboxes (libmpi.so.1.5.2: btl_vader_fbox.h,106)
1420000000	2.6965	unpack_predefined_data (libopen-pal.so.6.2.0: opal_datatype_unpack.h,41)
1220000000	2.3167	hypr_SemiInterp (smg2000: semi_interp.c,126)
1140000000	2.1648	pack_predefined_data (libopen-pal.so.6.2.0: opal_datatype_pack.h,38)
1020000000	1.9370	__memcpy_sse3_back (libc-2.17.so)
740000000	1.4052	hypr_SemiRestrict (smg2000: semi_restrict.c,125)
...		
...		

```
openss>>expview -v statements
```

Exclusive PAPI_TOT_CYC Counts	% of Total PAPI_TOT_CYC Counts	Statement Location (Line Number)
17800000000	36.9141	smg_residual.c(289)
3440000000	7.1340	cyclic_reduction.c(1130)
2780000000	5.7652	smg_residual.c(238)
2760000000	5.7238	cyclic_reduction.c(910)
1700000000	3.5255	cyclic_reduction.c(999)
1660000000	3.4426	btl_vader_fbox.h(119)
1180000000	2.4471	smg_residual.c(287)
960000000	1.9909	cyclic_reduction.c(853)
...		
...		

```
openss>>expview -v linkedobjects
```

Exclusive PAPI_TOT_CYC Counts	% of Total PAPI_TOT_CYC Counts	LinkedObject
40800000000	77.3606	smg2000
6060000000	11.4903	libmpi.so.1.5.2
4160000000	7.8878	libopen-pal.so.6.2.0
1720000000	3.2613	libc-2.17.so

5.2 Hardware Counter Time Experiment (hwctime)

In this example, the `osshwc` experiment runs on our test program, `SMG2000`. The convenience script for this experiment is:

```
> osshwctime "mpirun -np 256 smg2000 -n 50 50 50" <counter> <threshold>
```

This is the same syntax as the `osshwc` experiment. Note: If the output is empty, try lowering the `<threshold>` value; OJSS calculates it by default. Users can try lowering the threshold value if there have not been enough PAPI event occurrences to record. Also, see the hint in the `osshwcsamp` section below. Users can run `osshwcsamp` and use a formula to create a reasonable threshold. Any counter reported by “`papi_avail -a`” that is not derived is available for use. Users also can use the `osshwc` or `osshwctime` commands with no arguments to see the available counters. Native counters are listed in the PAPI documentation.

PAPI Name	Description	Threshold
PAPI_L1_DCM	L1 data cache misses	high
PAPI_L2_DCM	L2 data cache misses	high/medium
PAPI_L1_DCA	L1 data cache accesses	high
PAPI_FPU_IDL	Cycles in which FPUs are idle	high/medium
PAPI_STL_ICY	Cycles with no instruction issue	high/medium
PAPI_BR_MSP	Miss-predicted branches	medium/low
PAPI_FP_INS	Number of floating point instructions	high
PAPI_LD_INS	Number of load instructions	high
PAPI_VEC_INS	Number of vector/SIMD instructions	high/medium
PAPI_HW_INT	Number of hardware interrupts	low
PAPI_TLB_TL	Number of TLB misses	low

Note: Threshold indications are just for rough guidance and are dependent on the application. Also, remember that not all counters will exist on all platforms: Run `osshwc` with no arguments to see the available hardware counters.

The sections below show outputs from the `osshwctime` experiment. Note that the default counter is the total cycles.

5.2.1 Hardware Counter Time Threshold (hwctime) experiment performance data gathering

The hardware counter threshold experiment convenience script is “osshwc”. Here’s how to use it to gather counter values for one unique hardware counter:

```
osshwctime “how you normally run your application” <papi event > < threshold value>
```

The following example shows how to use the osshwc convenience script to gather data for the SMG2000 application on a Linux cluster platform. If there is no hardware counter value specified after the quoted application run command, the osshwctime convenience script will gather performance data for the default counter, PAPI_TOT_CYC. This example specifies an alternative counter, PAPI_L1_DCM, and a specific threshold value, 750000. Each time the threshold value is reached, a sample will be taken and recorded. At program completion, an O|SS database file is created and users can view the performance data. A default report is shown as part of the O|SS convenience script (below).

```
$ osshwctime "mpirun -np 4 ./smg2000 -n 65 65 65" PAPI_L1_DCM 750000
[openss]: hwctime using default threshold: 750000.
[openss]: hwctime using user specified papi event: "PAPI_L1_DCM"
Creating topology file for frontend host localhost
Generated topology file: ./cbtfAutoTopology
Running hwctime collector.
Program: mpirun -np 4 ./smg2000 -n 65 65 65
Number of mrnet backends: 4
Topology file used: ./cbtfAutoTopology
executing mpi program: mpirun -np 4 cbtfrun --mpi --mrnet -c hwctime ./smg2000 -n 65 65 65
Running with these driver parameters:
(nx, ny, nz) = (65, 65, 65)
(Px, Py, Pz) = (4, 1, 1)
(bx, by, bz) = (1, 1, 1)
(cx, cy, cz) = (1.000000, 1.000000, 1.000000)
(n_pre, n_post) = (1, 1)
dim = 3
solver ID = 0
=====
Struct Interface:
=====
Struct Interface:
wall clock time = 0.024858 seconds
cpu clock time = 0.030000 seconds
=====
Setup phase times:
=====
SMG Setup:
wall clock time = 0.624002 seconds
cpu clock time = 0.620000 seconds
=====
Solve phase times:
```

```

=====
SMG Solve:
  wall clock time = 3.907005 seconds
  cpu clock time  = 3.870000 seconds

Iterations = 7
Final Relative Residual Norm = 1.760588e-07

All Threads are finished.
default view for ./smg2000-hwctime-4.openss
[openss]: The restored experiment identifier is: -x 1
Performance data spans 4.818158 seconds from 2016/11/11 11:12:31 to 2016/11/11 11:12:36

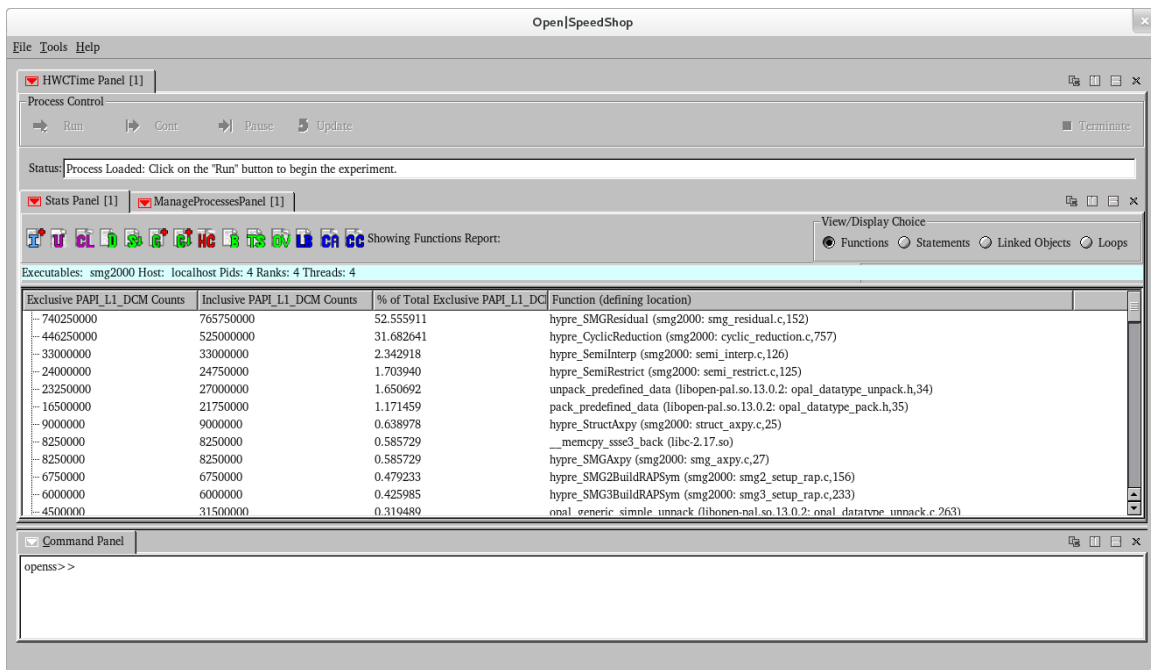
Exclusive Inclusive % of Total Function (defining location)
PAPI_L1_DCM PAPI_L1_DCM Exclusive
Counts Counts PAPI_L1_DCM
Counts
740250000 765750000 52.555911 hypre_SMGResidual (smg2000: smg_residual.c,152)
446250000 525000000 31.682641 hypre_CyclicReduction (smg2000: cyclic_reduction.c,757)
330000000 330000000 2.342918 hypre_SemiInterp (smg2000: semi_interp.c,126)
240000000 247500000 1.703940 hypre_SemiRestrict (smg2000: semi_restrict.c,125)
232500000 270000000 1.650692 unpack_predefined_data (libopen-pal.so.13.0.2:
opal_datatype_unpack.h,34)
165000000 217500000 1.171459 pack_predefined_data (libopen-pal.so.13.0.2:
opal_datatype_pack.h,35)
90000000 90000000 0.638978 hypre_StructAxy (smg2000: struct_axpy.c,25)
82500000 82500000 0.585729 hypre_SMGAxy (smg2000: smg_axpy.c,27)
82500000 82500000 0.585729 __memcpy_ssse3_back (libc-2.17.so)
67500000 67500000 0.479233 hypre_SMG2BuildRAPSym (smg2000: smg2_setup_rap.c,156)
60000000 60000000 0.425985 hypre_SMG3BuildRAPSym (smg2000: smg3_setup_rap.c,233)

```

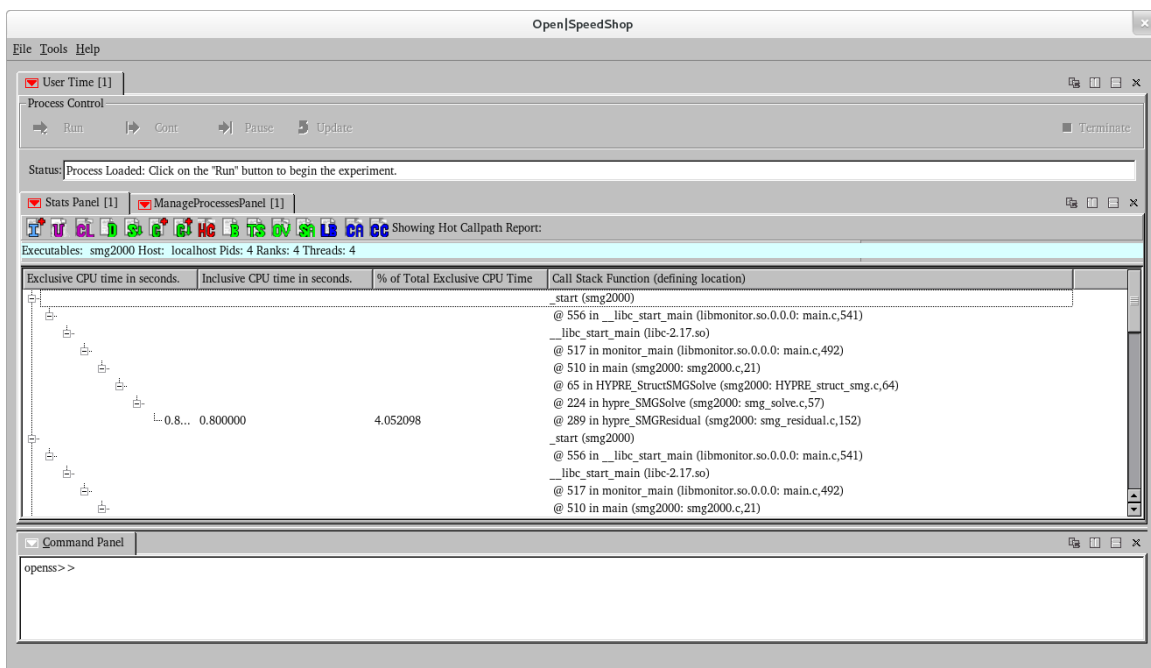
5.2.2 Viewing Hardware Counter Threshold (hwctime) experiment performance data via GUI

To launch the GUI on any experiment, use “openss -f <database name>”.

This image shows the default view for the hwc experiment run with the SMG2000 MPI application specifying PAPI_L1_DCM as the hardware counter event. Double-clicking on a performance information line in the Stats Panel or on the bar chart will take the user to the source file and line it represents.

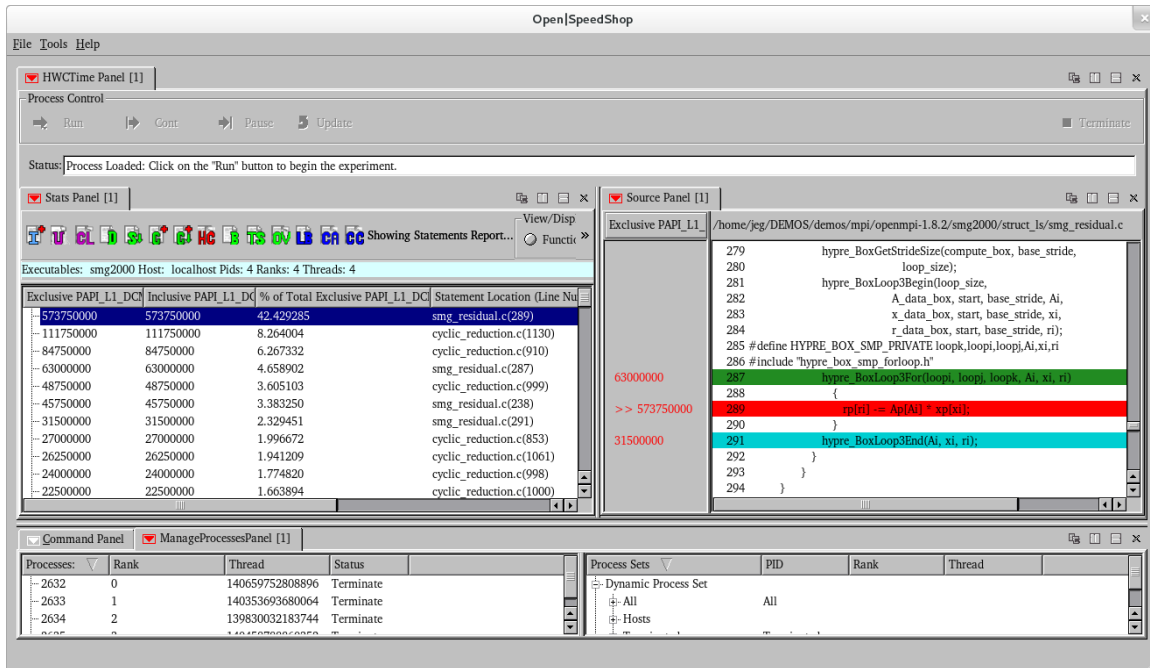


The next image displays output from the `osshwtime` experiment with the Hot Call Path icon (red HC) chosen. This displays the top five time-consuming call paths in the SMG2000 application.



The view below shows the top time-consuming statements, with the source panel focused on the one in SMG2000 that took the most. In the StatsPanel, double-

clicking on a statistics line will focus the source panel on the corresponding source line.



5.2.3 Viewing Hardware Counter Time Threshold (hwtime) experiment performance data via CLI

To launch the CLI on any experiment, use “openss -cli -f <database name>”. This example shows three default CLI views of varying granularities: function, statement and library level.

This is the CLI default view for the hwtime experiment:

```
$ openss -cli -f smg2000-hwtime-4.openss
openss>>[openss]: The restored experiment identifier is: -x 1
openss>>expview hwtime10
```

Exclusive PAPI_L1_DCM Counts	Inclusive PAPI_L1_DCM Counts	% of Total PAPI_L1_DCM	Function (defining location)
740250000	765750000	52.555911	hypre_SMGResidual (smg2000: smg_residual.c,152)
446250000	525000000	31.682641	hypre_CyclicReduction (smg2000: cyclic_reduction.c,757)
330000000	330000000	2.342918	hypre_SemiInterp (smg2000: semi_interp.c,126)
240000000	247500000	1.703940	hypre_SemiRestrict (smg2000: semi_restrict.c,125)
232500000	270000000	1.650692	unpack_predefined_data (libopen-pal.so.13.0.2: opal_datatype_unpack.h,34)
165000000	217500000	1.171459	pack_predefined_data (libopen-pal.so.13.0.2: opal_datatype_pack.h,35)
90000000	90000000	0.638978	hypre_StructAxy (smg2000: struct_axpy.c,25)
82500000	82500000	0.585729	hypre_SMGAxy (smg2000: smg_axpy.c,27)

8250000	8250000	0.585729	__memcpy_sse3_back (libc-2.17.so)
6750000	6750000	0.479233	hypre_SMG2BuildRAPSym (smg2000: smg2_setup_rap.c,156)

This CLI view for the hwtime experiment shows performance information based on loops in SMG2000:

```
openss>>expview -v loops hwtime10
```

Exclusive	Inclusive	% of Total	Loop Start Location (Line Number)
PAPI_L1_DCM	PAPI_L1_DCM	Exclusive	
Counts	Counts	PAPI_L1_DCM	
	Counts		
739500000	765000000	42.813721	smg_residual.c(204)
213000000	256500000	12.331741	cyclic_reduction.c(882)
192000000	227250000	11.115936	cyclic_reduction.c(1022)
41250000	41250000	2.388189	cyclic_reduction.c(851)
41250000	41250000	2.388189	cyclic_reduction.c(835)
40500000	40500000	2.344768	cyclic_reduction.c(851)
39000000	39000000	2.257924	cyclic_reduction.c(851)
24000000	24750000	1.389492	semi_restrict.c(198)
20250000	20250000	1.172384	semi_interp.c(292)
20250000	20250000	1.172384	semi_interp.c(292)

This CLI view for the hwtime experiment shows performance information based on statements in SMG2000. In this experiment, statement 289 had the most level 1 data cache misses:

```
openss>>expview -v statements hwtime10
```

Exclusive	Inclusive	% of Total	Statement Location (Line Number)
PAPI_L1_DCM	PAPI_L1_DCM	Exclusive	
Counts	Counts	PAPI_L1_DCM	
	Counts		
573750000	573750000	42.429285	smg_residual.c(289)
111750000	111750000	8.264004	cyclic_reduction.c(1130)
84750000	84750000	6.267332	cyclic_reduction.c(910)
63000000	63000000	4.658902	smg_residual.c(287)
48750000	48750000	3.605103	cyclic_reduction.c(999)
45750000	45750000	3.383250	smg_residual.c(238)
31500000	31500000	2.329451	smg_residual.c(291)
27000000	27000000	1.996672	cyclic_reduction.c(853)
26250000	26250000	1.941209	cyclic_reduction.c(1061)
24000000	24000000	1.774820	cyclic_reduction.c(998)

This CLI view for the hwtime experiment shows performance information based on libraries or linked objects in SMG2000. In this experiment, the executable had 92 percent of the level 1 data cache misses:

```
openss>>expview -v linkedobjects
```

Exclusive	Inclusive	% of Total	LinkedObject
PAPI_L1_DCM	PAPI_L1_DCM	Exclusive	
Counts	Counts	PAPI_L1_DCM	

Counts			
1297500000	1410000000	92.021277	smg2000
58500000	81750000	4.148936	libopen-pal.so.13.0.2
34500000	101250000	2.446809	libmpi.so.12.0.2
19500000	1410000000	1.382979	libc-2.17.so

5.3 Hardware Counter Sampling (hwcsamp) Experiment

The osshwcsamp experiment supports both derived and non-derived PAPI presets and can sample up to six counters simultaneously. Again, users can run osshwcsamp with no arguments to check the available counters. All native events are available, including architecture-specific events listed in the PAPI documentation. Native events also are reported by papi_native_avail.

The hardware counter sampling experiment uses a sampling rate rather than the threshold used in previous experiments. Like the threshold, however, the sampling rate depends on the application and users must strike a balance between overhead and accuracy. In this case, the lower the sampling rate, the fewer samples recorded.

The convenience script for this experiment is:

```
> osshwcsamp "mpirun -np 256 smg2000 -n 50 50 50" <event_list> <sampling_rate>
```

Note: If a counter does not appear in the output, there may be a conflict in the hardware counters. To find conflicts use:

```
> papi_event_chooser PRESET <list_of_events>
```

Here is a list (from Koushik Ghosh of LLNL) of some possible hardware counter combinations:

For Xeon processors:	
PAPI_FP_INS, PAPI_LD_INS, PAPI_SR_INS	Load store info, memory bandwidth needs
PAPI_L1_DCM, PAPI_L1_TCA	L1 cache hit/miss ratios
PAPI_L2_DCM, PAPI_L2_TCA	L2 cache hit/miss ratios
LAST_LEVEL_CACHE_MISSES, LAST_LEVEL_CACHE_REFERENCES	L3 cache info
MEM_UNCORE_RETIRED:REMOTE_DRAM, MEM_UNCORE_RETIRED:LOCAL_DRAM	Local/nonlocal memory access
For Opteron processors:	
PAPI_FAD_INS, PAPI_FML_INS	Floating point add multiply
PAPI_FDV_INS, PAPI_FSQ_INS	Square root and divisions
PAPI_DP_OPS, PAPI_VEC_INS	Floating point and vector instructions
READ_REQUEST_TO_L3_CACHE:ALL_CORES, L3_CACHE_MISSES:ALL_CORES	L3 cache

When selecting PAPI events, users must determine if they are a valid combination. In general, valid combinations will pass the test:

```
> papi_event_choser PRESET event1 event2 ... eventN
```

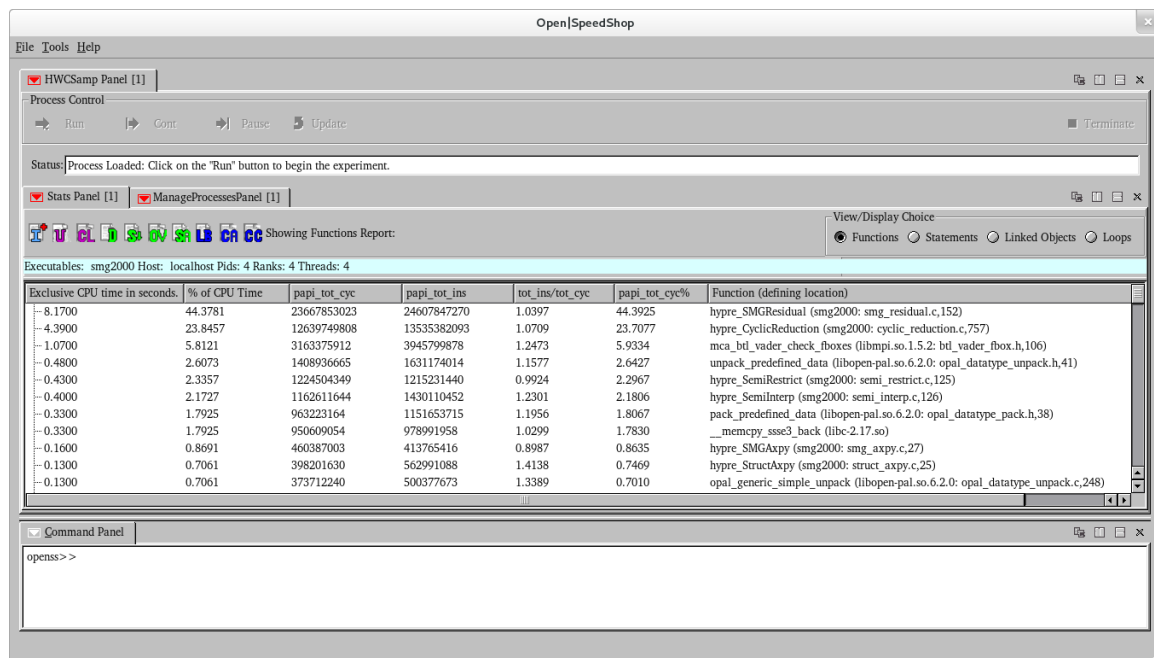
The output for a valid combination will contain:

```
event_choser.c PASSED
```

Here is an example of using PAPI to check the validity of a three-event combination:

```
> papi_event_choser PRESET PAPI_FP_INS PAPI_LD_INS PAPI_SR_INS
-----PAPI Version :4.1.2.1
Vendor string and code : GenuineIntel (1)
Model string and code : Intel Nehalem (21)
CPU Revision : 5.000000
...
PAPI_VEC_SP 0x80000069 No Single precision vector/SIMD instructions
PAPI_VEC_DP 0x8000006a No Double precision vector/SIMD instructions
-----Total events reported: 44
event_choser.c PASSED
```

Here is the osshwcsamp experiment output with counters for total cycles and floating point operations:



The screenshot shows the OpenSpeedShop GUI with the HWCSamp experiment running. The 'Stats Panel' displays a table of execution statistics for various functions. The table has columns for Exclusive CPU time in seconds, % of CPU Time, papi_tot_cyc, papi_tot_ins, tot_ins/tot_cyc, papi_tot_cyc%, and Function (defining location).

Exclusive CPU time in seconds	% of CPU Time	papi_tot_cyc	papi_tot_ins	tot_ins/tot_cyc	papi_tot_cyc%	Function (defining location)
8.1700	44.3781	23667853023	24607847270	1.0397	44.3925	hypr_SMGResidual (smg2000: smg_residual.c,152)
4.3900	23.8457	12639749808	13535382093	1.0709	23.7077	hypr_CyclicReduction (smg2000: cyclic_reduction.c,757)
1.0700	5.8121	3163375912	3945799878	1.2473	5.9334	mca_btl_vader_check_fboxes (libmpi.so.1.5.2: btl_vader_fbox.h,106)
0.4800	2.6073	1408936665	1631174014	1.1577	2.6427	unpack_predefined_data (libopen-pal.so.6.2.0: opal_datatype_unpack.h,41)
0.4300	2.3357	1224504349	1215231440	0.9924	2.2967	hypr_SemiRestrict (smg2000: semi_restrict.c,125)
0.4000	2.1727	1162611644	1430110452	1.2301	2.1806	hypr_SemiInterp (smg2000: semi_interp.c,126)
0.3300	1.7925	963223164	1151653715	1.1956	1.8067	pack_predefined_data (libopen-pal.so.6.2.0: opal_datatype_pack.h,38)
0.3300	1.7925	950609054	978991958	1.0299	1.7830	_memcpy_sse3_back (libc-2.17.so)
0.1600	0.8691	460387003	413765416	0.8987	0.8635	hypr_SMGXpy (smg2000: smg_axpy.c,27)
0.1300	0.7061	398201630	562991088	1.4138	0.7469	hypr_StructXpy (smg2000: struct_axpy.c,25)
0.1300	0.7061	373712240	500377673	1.3389	0.7010	opal_generic_simple_unpack (libopen-pal.so.6.2.0: opal_datatype_unpack.c,248)

Remember: It's not always necessary to use the OJSS GUI to examine experiment output; the command line interface is available to view the same information. For example, the output from above can be seen on the command line:

```
>openss -cli -f smg2000-hwcsamp.openss
openss>>[openss]: The restored experiment identifier is: -x 1
openss>>expview
Exclusive % of CPU papi_tot_cyc papi_tot_ins tot_ins/tot_cyc papi_tot_cyc% Function (defining location)
```

CPU time	Time				IPC		
8.1700	44.3781	23667853023	24607847270	1.0397	44.3925	hypre_SMGResidual (smg2000: smg_residual.c,152)	
4.3900	23.8457	12639749808	13535382093	1.0709	23.7077	hypre_CyclicReduction (smg2000: cyclic_reduction.c,757)	
1.0700	5.8121	3163375912	3945799878	1.2473	5.9334	mca_btl_vader_check_fboxes (libmpi.so.1.5.2: btl_vader_fbox.h,106)	
0.4800	2.6073	1408936665	1631174014	1.1577	2.6427	unpack_predefined_data (libopen-pal.so.6.2.0: opal_datatype_unpack.h,41)	
0.4300	2.3357	1224504349	1215231440	0.9924	2.2967	hypre_SemiRestrict (smg2000: semi_restrict.c,125)	
0.4000	2.1727	1162611644	1430110452	1.2301	2.1806	hypre_SemiInterp (smg2000: semi_interp.c,126)	
openss>>expview -v linkedobjects							
Exclusive CPU time	% of CPU Time	papi_tot_cyc	papi_tot_ins	tot_ins/tot_cyc	papi_tot_cyc%	LinkedObject	IPC
14.4400	78.3931	41735382047	44436967182	1.0647	78.2394	smg2000	
1.9400	10.5320	5687487004	7106186325	1.2494	10.6621	libmpi.so.1.5.2	
1.3400	7.2747	3918861297	4788179789	1.2218	7.3465	libopen-pal.so.6.2.0	
0.6700	3.6374	1918417268	2138815631	1.1149	3.5964	libc-2.17.so	
0.0300	0.1629	83014429	77541115	0.9341	0.1556	libpthread-2.17.so	
18.4200	100.0000	53343162045	58547690042	1.0976	100.0000	Report Summary	

5.3.1 Hardware Counter Sampling (hwcsamp) experiment performance data gathering

The hardware counter sampling experiment convenience script is “osshwcsamp”. Here’s how to use this to gather values for up to six unique hardware counters:

```
osshwcsamp “how you normally run your application” <papi event list> <sampling rate>
```

5.3.1.1 Hardware Counter Sampling (hwcsamp) experiment parameters

The hwcsamp experiment is timer-based, not threshold-based: A timer periodically interrupts the processor. For the hwcsamp experiment, each time that happens the values of the specified hardware counter events will be read up and reset to 0 for the next timer cycle. This is repeated until the program finishes. O|SS lets the user control the sampling rate.

Here’s an example of how to gather data for the SMG2000 application on a Linux cluster platform using the osshwcsamp convenience script and specifying a set of PAPI hwc events. In the second example, the user chooses to sample only 45 times a second instead of the default 100 times. Users may do this to save database size, as a lower sampling rate may accurately portray the application behavior.

```
> osshwcsamp “mpirun -np 256 smg2000 -n 50 50 50” PAPI_L1_DCM,PAPI_L2_DCA,PAPI_L2_DCM,PAPI_L3_DCA,PAPI_L3_TCM
```

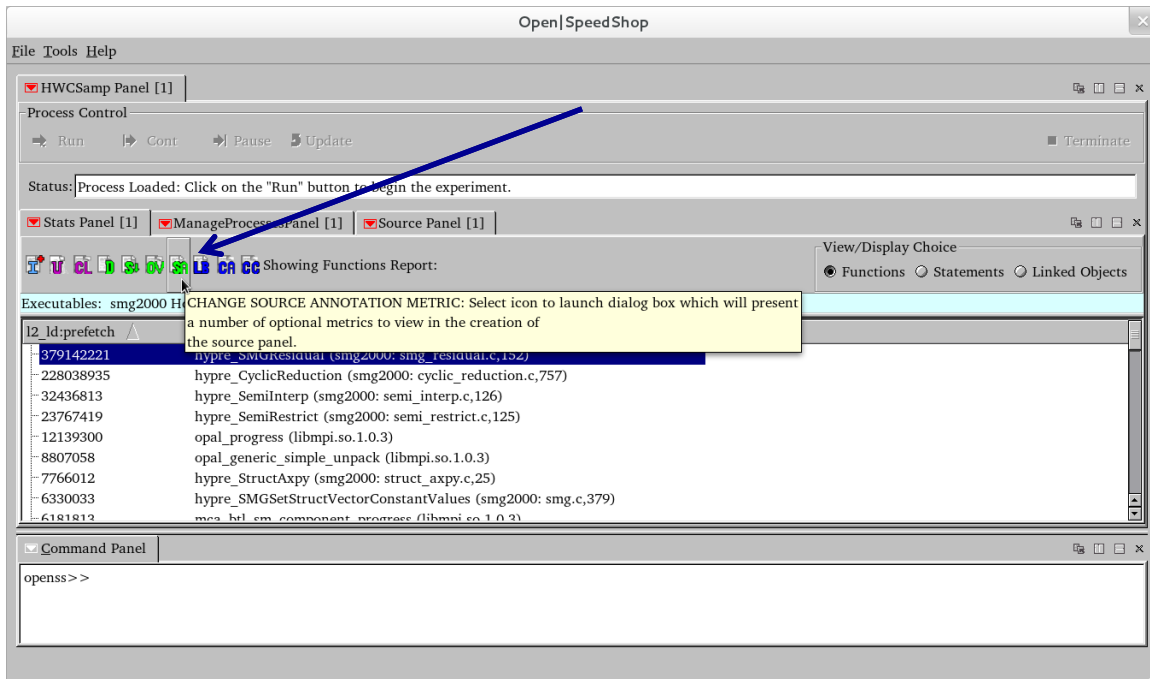
```
> osshwcsamp “mpirun -np 256 smg2000 -n 50 50 50” PAPI_L1_DCM,PAPI_L2_DCA,PAPI_L2_DCM 45
```

5.3.2 Viewing Hardware Counter Sampling (hwcsamp) experiment performance data via GUI

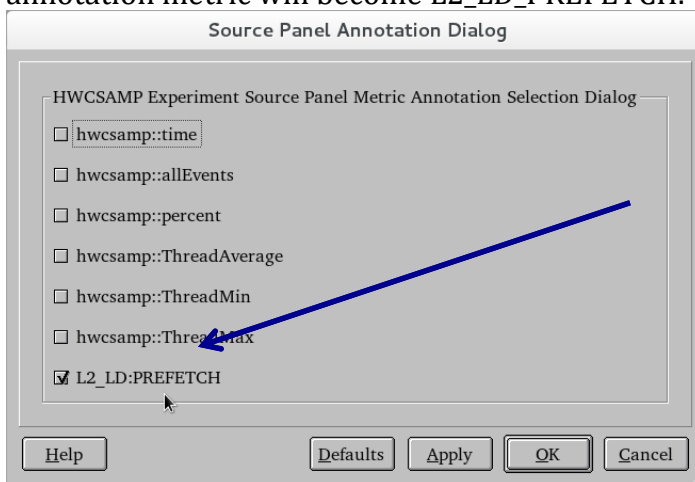
To launch the GUI on any experiment, use “openss -f <database name>”.

5.3.2.1 Getting the PAPI counter as the GUI Source Annotation Metric

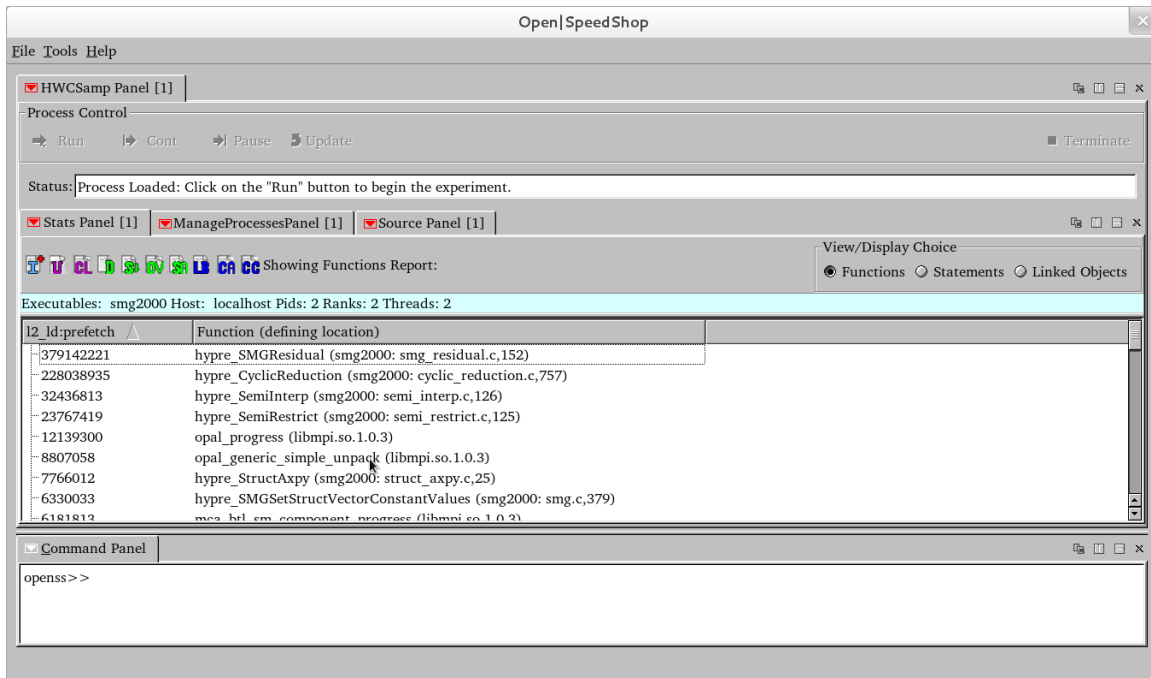
To make one of the PAPI or native hardware counters the counter that will appear in the source view, click on the SA (Source Annotation) icon. This opens an option dialogue that allows users to choose the source annotation metric.



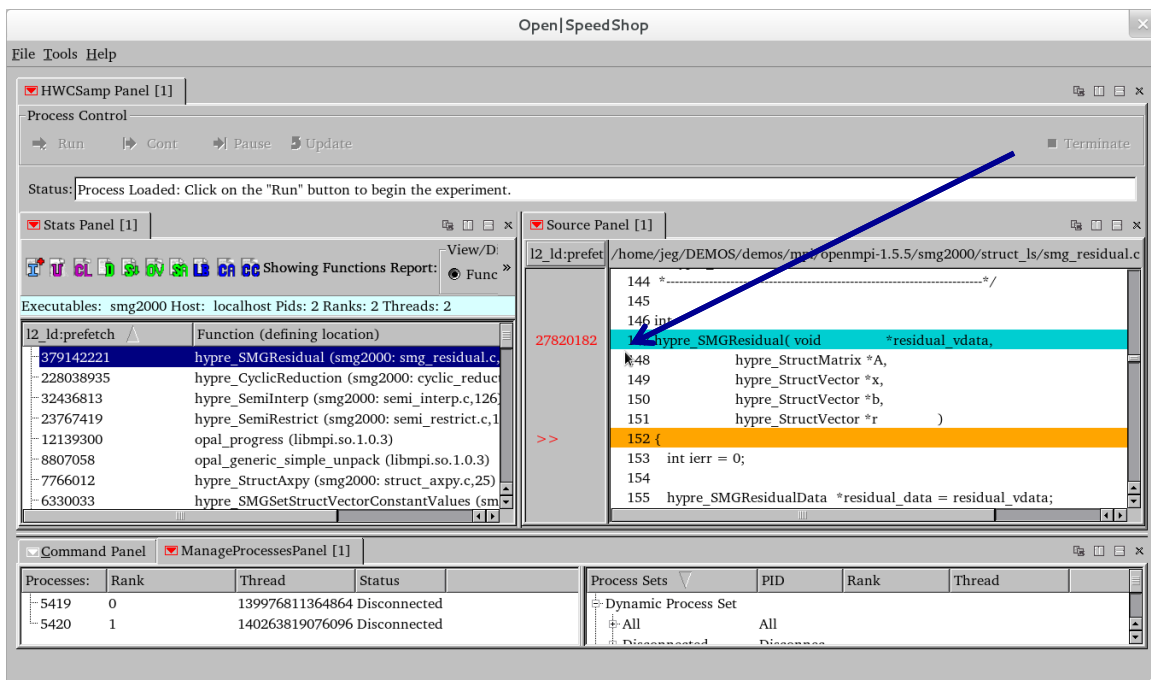
In this example, the L2_LD_PREFETCH native counter is chosen. When users choose that counter and click OK, the Stats Panel view will regenerate and the source annotation metric will become L2_LD_PREFETCH.



The regenerated view now shows results for only L2_LD_PREFETCH.



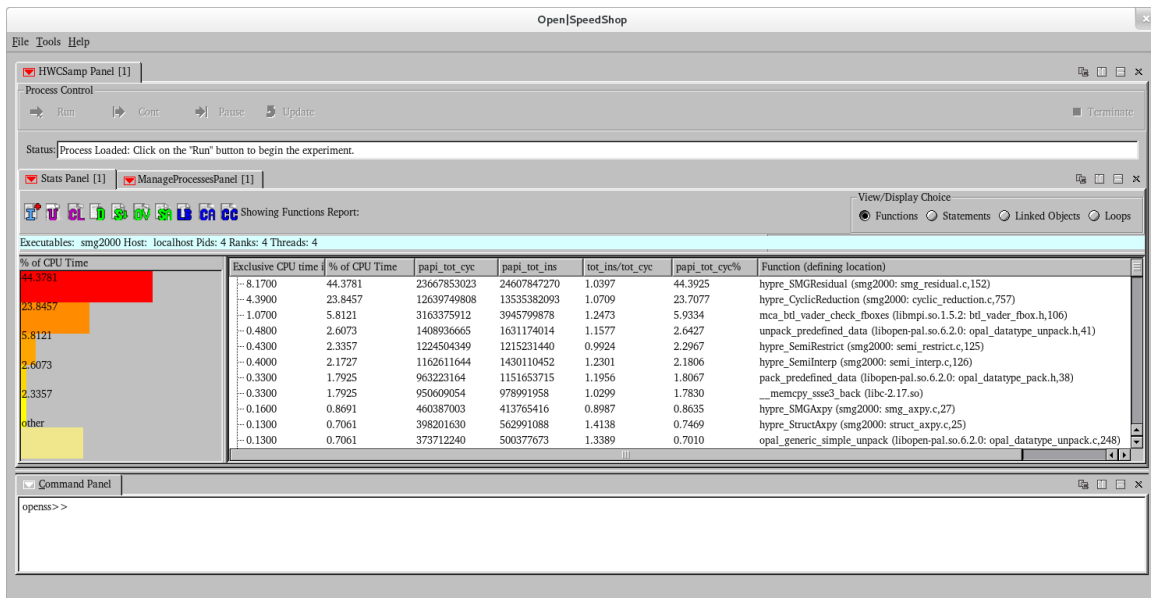
Double-clicking on a particular Stats Panel line will focus the source panel and use the PAPI or native counter that was chosen via the Source Annotation dialog.



5.3.2.2 Viewing Hardware Counter Sampling Data via GUI

To launch the GUI on any experiment, use "openss -f <database name>".

The default GUI view of a hardware counter sampling (hwcsamp) experiment example is below. The first set of performance data shown is program counter exclusive time (where the program is statistically spending its time) and the percentage of time spent in each program function. Next are the hardware counter event counts listed in columns: Column three shows counts recorded for PAPI_TOT_CYC; column four shows event counts for PAPI_TOT_INS. This view can indicate whether the specified hardware counter events are occurring and, if they are, their prevalence. With this information, users can use the hwc or hwctime experiment to isolate exactly where a particular event is occurring. These two experiments (hwc and hwctime) are threshold-based: because the actual event triggered recording the event counts, users can map performance data back to the source. The hwcsamp experiment is timer-based, so O|SS cannot take users to the exact line of source where the hardware counter events are happening. It is an overview experiment that tells users which events are occurring. It also tells whether events are occurring in numbers that would warrant using the hwc or hwctime experiments to pinpoint the source location where the specified hardware counter event actually occurs.



5.3.3 Viewing Hardware Counter Sampling (hwcsamp) experiment performance data via CLI

To launch the CLI on any experiment, use “openss -cli -f <database name>”. This example was run on the Yellowstone platform at NCAR/UCAR using the job script shown below.

5.3.3.1 Job Script and osshwcsamp command

```
#!/bin/csh
#
# LSF batch script to run an MPI application
#
#BSUB -P Pnnnnnnnnn      # project code
#BSUB -W 00:30           # wall-clock time (hrs:mins)
#BSUB -n 64              # number of tasks in job
#BSUB -R "span[ptile=4]"  # run 4 MPI tasks per node
#BSUB -j sweep3d-hwcsamp   # job name
#BSUB -o sweep3d-hwcsamp.%J.out  # output file name in which %J is replaced by the job ID
#BSUB -e sweep3d-hwcsamp.%J.err  # error file name in which %J is replaced by the job ID
#BSUB -q regular          # queue

module load openspeedshop

mkdir -p /glade/scratch/${USER}/sweep3d
rm -rf /glade/scratch/${USER}/sweep3d/hwcsamp
mkdir /glade/scratch/${USER}/sweep3d/hwcsamp
setenv OPENSS_RAWDATA_DIR /glade/scratch/${USER}/sweep3d/hwcsamp

setenv REQUEST_SUSPEND_HPC_STAT 1

echo "running (on compute node): osshwcsamp"
osshwcsamp "mpirun.lsf /glade/u/home/galaro/demos/sweep3d/orig/sweep3d.mpi"
PAPI_L1_DCM,PAPI_L1_ICM,PAPI_L1_TCM,PAPI_L1_LDM,PAPI_L1_STM
```

5.3.3.2 osshwcsamp experiment default CLI view

This table describes information included in the hwcsamp experiment default view when no alternative PAPI hardware counter arguments are specified.

Column Name	Column Definition
Exclusive CPU Time	Aggregated total exclusive time spent in the application function corresponding to this row of data.
% of CPU Time	Percentage of exclusive time spent in the function corresponding to this row of data relative to the total application exclusive time for all the application functions.
PAPI_TOT_CYC	Number of hardware events corresponding to the hardware independent PAPI_TOT_CYC PAPI event. This value is based on reading the hardware counter event buffers using sampling. This means this data may not accurately reflect where in the source these events occurred. It is an approximation of what is going in the application, but does not map back to the source lines. Use the hwc and hwctime experiments for that.
PAPI_TOT_INS	Number of hardware events corresponding to the hardware independent PAPI_TOT_INS PAPI event. This value is based on reading the hardware counter event buffers using sampling. This means this data may not accurately reflect where these events occurred in the source. It is an approximation of what is going in the application, but does not map back to the source lines. Use the hwc and hwctime experiments for that.

Column Name	Column Definition
TOT_INS/TOT_CYC	This is the graduated instructions per cycle, which is the ratio between the approximation of the total number of instructions divided by the total number of cycles
% of TOT_CYC	The percentage of PAPI_TOT_CYC events for this function relative to the number of PAPI_TOT_CYC events that occurred in all the application functions.

This is a default CLI view for the hwcsamp experiment:

```
Exclusive % of CPU papi_tot_cyc papi_tot_ins tot_ins/tot_cyc papi_tot_cyc% Function (defining location)
CPU time Time
in
seconds.
74.0600 99.8786 177712237021 51989184616 0.2925 99.8787 main (nbody: nbody-mpi.c,71)
0.0400 0.0539 95958566 28058948 0.2924 0.0539 fesetenv (libm-2.19.so)
0.0300 0.0405 71987793 21053819 0.2925 0.0405 __sqrt_finite (libm-2.19.so)
0.0100 0.0135 23864331 6996727 0.2932 0.0134 memcpy (libc-2.19.so)
0.0100 0.0135 23995616 7018006 0.2925 0.0135 fegetround (libm-2.19.so)
74.1500 100.0000 177928043327 52052312116 0.2925 100.0000 Report Summary
```

This is the output from a non-default osshwcsamp experiment which specified PAPI_L1_DCM,PAPI_L1_ICM,PAPI_L1_TCM,PAPI_L1_LDM,PAPI_L1_STM on the osshwcsamp command:

```
openss -cli -f L1-64PE-sweep3d.mpi-hwcsamp.openss
openss>>[openss]: The restored experiment identifier is: -x 1
openss>>expview -v summary

Exclusive % of CPU papi_l1_dcm papi_l1_icm papi_l1_tcm papi_l1_ldm papi_l1_stm Function (defining location)
CPU time in Time
seconds.
824.870000 38.689781 8646497071 117738843 8764235914 8396159476 196649065 __libc_poll (libc-2.12.so)
799.300000 37.490443 46691996441 367096209 47059092650 46247555479 281624221 sweep (sweep3d.mpi:
sweep.f,2)
75.000000 3.517807 782716992 10680760 793397752 757322217 20159725
PAMI::Interface::Context<PAMI::Context>::advance (libpami.so: ContextInterface.h,158)
55.750000 2.614903 597583047 8038242 605621289
579127274 14647999 LapiImpl::Context::Advance<true, true, false> (libpami.so: Context.h,220)
52.970000 2.484510 550761926 7569975 558331901 535841812 11563657 __libc_enable_asynccancel (libc-
2.12.so)
49.850000 2.338169 518605433 6979361 525584794 502551336 12757207 _lapi_dispatcher<false> (libpami.so:
lapi_dispatcher.c,57)
48.080000 2.255149 488545916 6784192 495330108 476065093 9649598 LapiImpl::Context::TryLock<true, true,
false> (libpami.so: Context.h,198)
47.750000 2.239671 479947719 6732551 486680270 471343480 6436257 __libc_disable_asynccancel (libc-
2.12.so)
26.680000 1.251401 275998769 3888499 279887268 269841454 4697170 udp_read_callback (libpamiudp.so:
lapi_udp.c,538)
25.880000 1.213878 1522697263 12118336 1534815599 1507685061 9619348 __intel_sse3_rep_memcpy
(libirc.so)
21.960000 1.030014 223197680 3086626 226284306 215787794 5879517 _lapi_shm_dispatcher (libpami.so:
lapi_shm.c,2283)
14.910000 0.699340 154744623 2075688 156820311 149803306 3979337 LapiImpl::Context::CheckContext
(libpami.so: CheckParam.cpp,21)
13.990000 0.656188 151052863 2000330 153053193 146967548 3167039 LapiImpl::Context::Unlock<true, true,
false> (libpami.so: Context.h,204)
```


5.3.3.2 osshwcsamp experiment Status command and CLI view

```
openss>>expstatus
```

```
Experiment definition
{ # Expld is 1, Status is NonExistent, Saved database is L1-64PE-sweep3d.mpi-hwcsamp.openss
  Performance data spans 1:7.958138 mm:ss from 2013/03/27 22:32:45 to 2013/03/27 22:33:53
  Executables Involved:
    sweep3d.mpi
  Currently Specified Components:
    -h ys6128 -p 2765 -t 47176895393312 -r 3 (sweep3d.mpi)
    -h ys6128 -p 2766 -t 47824321252896 -r 0 (sweep3d.mpi)
    -h ys6128 -p 2767 -t 47369830317600 -r 1 (sweep3d.mpi)
    -h ys6128 -p 2768 -t 47378742910496 -r 2 (sweep3d.mpi)
    -h ys6129 -p 22862 -t 47327259860512 -r 5 (sweep3d.mpi)
    -h ys6129 -p 22863 -t 47201888194080 -r 6 (sweep3d.mpi)
    -h ys6129 -p 22864 -t 47185544437280 -r 7 (sweep3d.mpi)
    ...
    -h ys6250 -p 11462 -t 47028080107040 -r 63 (sweep3d.mpi)
    -h ys6250 -p 11463 -t 47600632852000 -r 60 (sweep3d.mpi)
    -h ys6250 -p 11464 -t 47494028697120 -r 61 (sweep3d.mpi)
    -h ys6250 -p 11465 -t 47944527175200 -r 62 (sweep3d.mpi)
  Previously Used Data Collectors:
    hwcsamp
  Metrics:
    hwcsamp::exclusive_detail
    hwcsamp::percent
    hwcsamp::threadAverage
    hwcsamp::threadMax
    hwcsamp::threadMin
    hwcsamp::time
  Parameter Values:
    hwcsamp::event = PAPI_L1_DCM,PAPI_L1_ICM,PAPI_L1_TCM,PAPI_L1_LDM,PAPI_L1_STM
    hwcsamp::sampling_rate = 100
  Available Views:
    hwcsamp
}
```

5.3.3.3 osshwcsamp experiment Load Balance command and CLI view

```
openss>>expview -m loadbalance
```

```
Max CPU Rank  Min CPU Rank  Average Function (defining location)
Time of CPU Time
Across Max Across Min Across
Ranks(s) Ranks(s) Ranks(s)
14.890000 28 10.950000 27 12.888594 _libc_poll (libc-2.12.so)
14.270000 47 11.780000 51 12.489062 sweep (sweep3d.mpi: sweep.f,2)
1.620000 43 0.840000 37 1.171875 PAPI::Interface::Context<PAPI::Context>::advance (libpami.so:
ContextInterface.h,158)
1.320000 16 0.570000 3 0.871094 LaplImpl::Context::Advance<true, true, false> (libpami.so: Context.h,220)
1.130000 60 0.500000 2 0.778906 _lapi_dispatcher<false> (libpami.so: lapi_dispatcher.c,57)
1.110000 35 0.520000 49 0.751250 LaplImpl::Context::TryLock<true, true, false> (libpami.so: Context.h,198)
1.030000 42 0.600000 12 0.827656 _libc_enable_asynccancel (libc-2.12.so)
0.950000 62 0.520000 38 0.746094 _libc_disable_asynccancel (libc-2.12.so)
0.700000 6 0.200000 59 0.343125 _lapi_shm_dispatcher (libpami.so: lapi_shm.c,2283)
0.630000 33 0.250000 0 0.404375 _intel_ssse3_rep_memcpy (libirc.so)
0.600000 18 0.270000 16 0.416875 udp_read_callback (libpamiudp.so:
```

5.3.3.4 osshwcsamp experiment Linked Object command and CLI view

```
openss>>expview -v linkedobjects
```

```
Exclusive % of CPU papi_l1_dcm papi_l1_icm papi_l1_tcm papi_l1_ldm papi_l1_stm LinkedObject
CPU time in Time
seconds.
```

```

928.310000 43.541541 9818946796 133244862 9952191658 9543597734 215608918 libc-2.12.so
811.920000 38.082373 47212355914 369525459 47581881373 46596204924 441601622 sweep3d.mpi
311.490000 14.610157 3356646038 44875637 3401521675 3255300343 80090932 libpami.so
29.640000 1.390237 1824778610 12931604 1837710214 1680978945 127174346 libirc.so
26.930000 1.263127 287313329 3994016 291307345 281053971 4763152 libpamiudp.so
22.250000 1.043616 1049603690 9037920 1058641610 1033650896 11422120 libpthread-2.12.so
1.440000 0.067542 72649683 620083 73269766 71327993 1007704 libmpich.so.3.3
0.020000 0.000938 1286256 23770 1310026 1232178 5222 ld-2.12.so
0.010000 0.000469 327 394 721 313 13 librt-2.12.so
2132.010000 100.000000 63623580643 574253745 64197834388 62463347297 881674029 Report Summary
openss>>

```

5.3.3.5 osshwcsamp experiment displaying only the hwcsamp PAPI events CLI view

The `-m allEvents` option prints only the PAPI event values and not the program counter sampling exclusive time and percentage values:

```

openss -cli -f L1-64PE-sweep3d.mpi-hwcsamp.openss
openss>>[openss]: The restored experiment identifier is: -x 1
openss>>expview -m allEvents

papi_l1_dcm papi_l1_icm papi_l1_tcm papi_l1_ldm papi_l1_stm Function (defining location)
8646497071 117738843 8764235914 8396159476 196649065 _libc_poll (libc-2.12.so)
46691996441 367096209 47059092650 46247555479 281624221 sweep (sweep3d.mpi: sweep.f,2)
782716992 10680760 793397752 757322217 20159725 PAMI::Interface::Context<PAMI::Context>::advance
597583047 8038242 605621289 579127274 14647999 LapiImpl::Context::Advance<true, true, false>
550761926 7569975 558331901 535841812 11563657 _libc_enable_asynccancel (libc-2.12.so)
518605433 6979361 525584794 502551336 12757207 _lapi_dispatcher<false> (libpami.so:
lapi_dispatcher.c,57)
488545916 6784192 495330108 476065093 9649598 LapiImpl::Context::TryLock<true, true, false>
479947719 6732551 486680270 471343480 6436257 _libc_disable_asynccancel (libc-2.12.so)
275998769 3888499 279887268 269841454 4697170 udp_read_callback (libpamiudp.so: lapi_udp.c,538)
1522697263 12118336 1534815599 1507685061 9619348 _intel_ssse3_rep_memcpy (libirc.so)
223197680 3086626 226284306 215787794 5879517 _lapi_shm_dispatcher (libpami.so: lapi_shm.c,2283)
154744623 2075688 156820311 149803306 3979337 LapiImpl::Context::CheckContext (libpami.so:
CheckParam.cpp,21)
151052863 2000330 153053193 146967548 3167039 LapiImpl::Context::Unlock<true, true, false>
(libpami.so: Context.h,204)

```

6 I/O Tracing and I/O Profiling

6.1 O|SS I/O Tracing General Usage

The O|SS `io` and `iot` I/O function-tracing experiments wrap the most common I/O functions, record the time spent in each, record the call path along which an I/O function was called, record the time spent along each call path to an I/O function, and record the number of times each function was called. In addition, the `iot` experiment also records information about each individual I/O function call. The values of the arguments and the return value from the I/O function are recorded.

6.2 I/O Base Tracing (`io`) experiment

This base I/O tracing experiment gathers data for these I/O functions: `close`, `creat`, `creat64`, `dup`, `dup2`, `lseek`, `lseek64`, `open`, `open64`, `pipe`, `pread`, `pread64`, `pwrite`, `pwrite64`, `read`, `readv`, `write` and `writv`. It is a trace-type experiment that wraps the

real I/O calls and records information before and after calling the real I/O functions. This I/O experiment records the basic I/O information as stated in the introductory section, but does not record the arguments to each call. The extended (iot) experiment does that.

6.2.1 I/O Base Tracing (io) experiment performance data gathering

The base I/O tracing (io) experiment convenience script is “ossio”. Here’s how to use this convenience script to gather base I/O tracing performance data:

```
ossio “how you normally run your application” <list of I/O function(s)>
```

Here’s an example of how to use the ossio convenience script to gather data for the IOP application on a Linux cluster platform. It gathers performance data for all the I/O functions because there are no list I/O functions specified after the quoted application run command:

```
ossio "srun -n 512 ./IOR"
```

6.2.2 Viewing I/O Base Tracing (io) experiment performance data via CLI

To launch the CLI on any experiment, use “openss -cli -f <database name>”.

6.2.3 Viewing I/O Base Tracing (io) experiment performance data via GUI

To launch the GUI on any experiment, use “openss -f <database name>”.

6.3 I/O Extended Tracing (iot) experiment

6.3.1 I/O Extended Tracing (iot) experiment performance data gathering

The extended I/O tracing (iot) experiment convenience script is “ossiott”. Here’s how to use this to gather extended I/O tracing performance data:

```
ossiott “how you normally run your application” <list of I/O function(s)>
```

Here’s an example of how to gather data for the IOP application on a Linux cluster platform using the ossiott convenience script. It gathers performance data for all the I/O functions because there are no list I/O functions specified after the quoted application run command:

```
ossiott "srun -n 512 ./IOR"
```

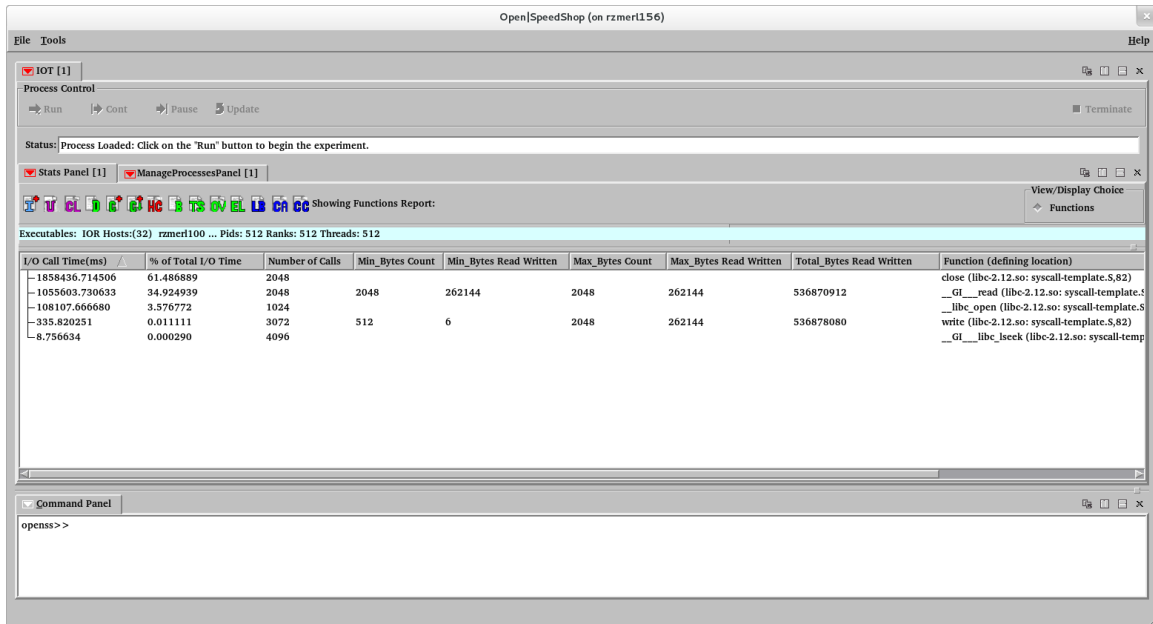
6.3.2 Viewing I/O Extended Tracing (iot) experiment performance data via GUI

To launch the GUI on any experiment, use “openss -f <database name>”.

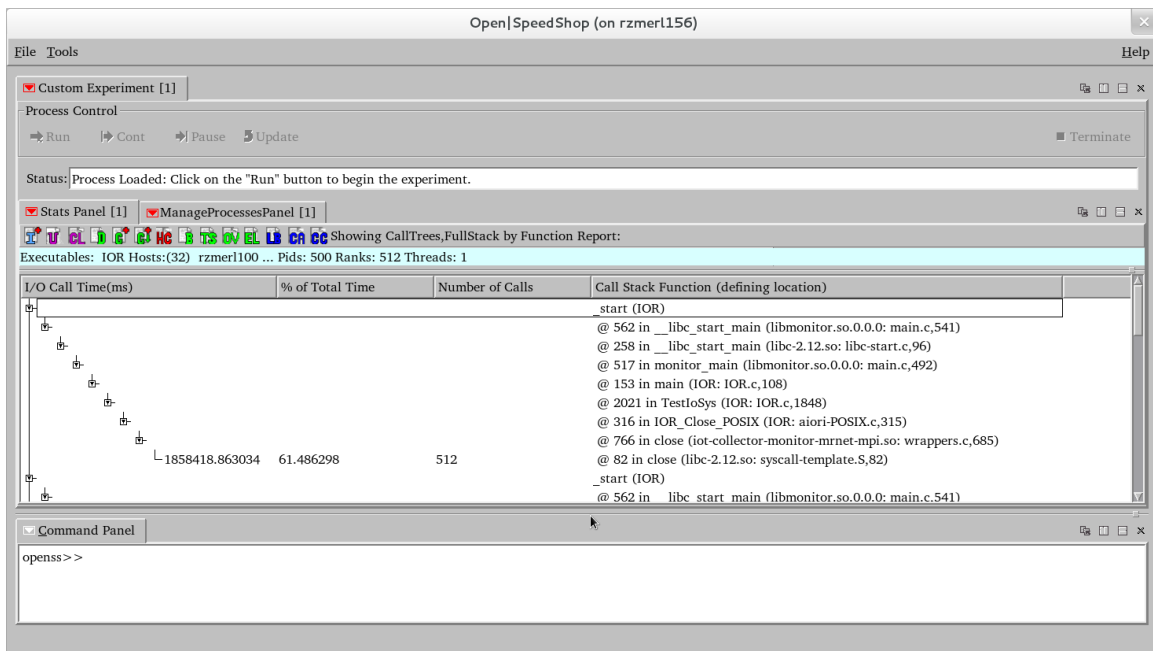
The default GUI view for the iot experiment is below. It summarizes the I/O functions that were called, how many times they were called and the time spent in each function. The percentage of the total I/O time also is attributed to each I/O function. The time is aggregated (totaled) across all the threads, ranks, or processes in the application. This table describes what the columns of data represent.

Choose one of the call path views to see functions that called the I/O functions.

Column Name	Column Definition
I/O Call Time	Aggregated total exclusive time spent in the I/O function corresponding to this row of data.
% of I/O Total Time	Percentage of exclusive time relative to the total time spent in the I/O function corresponding to this row of data.
Number of Calls	Total number of calls to the I/O function corresponding to this row of data.
Min Bytes Count	The number of times minimum bytes read or written by the corresponding I/O function occurred during this experiment.
Min Bytes Read or Written	The minimum number of bytes that were read or written by the corresponding I/O function.
Max Bytes Count	The number of times maximum bytes read or written by the corresponding I/O function occurred during this experiment.
Max Bytes Read or Written	The maximum number of bytes that were read or written by the corresponding I/O function.
Total Bytes Read or Written	The total number of bytes read or written by the corresponding function. This number only represents the totals for the number of bytes read or written based on the I/O function called.



Below, the user has chosen the C+ view icon. The Stats Panel now shows all the call paths in the user's application. This view shows every possible call path through the source to all the I/O functions called during execution. From this, the user could validate that this is expected behavior or find where the I/O is behaving unexpectedly.



Below is the load balance view, which provides the min, max and average values for the I/O function call time across all ranks in this application. This view shows some wide ranges between min and max values for some I/O functions. It may be valuable to use the Cluster Analysis view try to identify the ranks.

To launch the CLI on any experiment, use “openss -cli -f <database name>”.

The CLI can provide the same data options as the GUI views. Here are some examples of the performance data users can view and the commands to generate the CLI views. The following table describes the headers and meanings of the default iot view CLI columns.

Column Name	Column Definition
I/O Call Time	Aggregated total exclusive time spent in the I/O function corresponding to this row of data.
% of I/O Total Time	Percentage of exclusive time relative to the total time spent in the I/O function corresponding to this row of data.
Number of Calls	Total number of calls to the I/O function corresponding to this row of data.
Min Bytes Count	The number of times minimum bytes read or written by the corresponding I/O function occurred during this experiment.
Min Bytes Read or Written	The minimum number of bytes that were read or written by the corresponding I/O function.
Max Bytes Count	The number of times maximum bytes read or written by the corresponding I/O function occurred during this experiment.
Max Bytes Read or Written	The maximum number of bytes that were read or written by the corresponding I/O function.
Total Bytes Read or Written	The total number of bytes read or written by the corresponding function. This number only represents the totals for the number of bytes read or written based on the I/O function called.

>openss -cli -f IOR-iot.openss

openss>>[openss]: The restored experiment identifier is: -x 1

openss>>**expview**

I/O Call Time	% of Total Time(ms)	Number of I/O Calls	Min_Bytes Count	Min_Bytes Read Written	Max_Bytes Count	Max_Bytes Read Written	Total_Bytes Read Written	Function (defining location)
1858436.71	61.48	2048						close (libc-2.12.so)
1055603.73	34.92	2048	2048	262144	2048	262144	536870912	__GI__read (libc-2.12.so)
108107.66	3.57	1024						__libc_open (libc-2.12.so)
335.82	0.01	3072	512	6	2048	262144	536878080	write (libc-2.12.so)
8.75	0.003	4096						__GI__libc_lseek (libc-2.12.so)

Show load balance based on exclusive time spent in the I/O Functions

openss>>**expview -m loadbalance**

Max I/O Call Time Across Ranks(ms)	Rank of Max	Min I/O Call Time Across Ranks(ms)	Rank of Min	Average Call Time Across Ranks(ms)	I/O Function (defining location)
4114.522156	509	2680.653110	273	3629.759208	close (libc-2.12.so: syscall-template.S,82)
2824.349452	346	0.315392	317	2061.726036	__GI__read (libc-2.12.so: syscall-template.S,82)
989.579445	358	5.784552	414	211.147786	__libc_open (libc-2.12.so: syscall-template.S,82)
4.574762	65	0.424622	494	0.655899	write (libc-2.12.so: syscall-template.S,82)
0.044708	184	0.011079	317	0.017103	__GI__libc_lseek (libc-2.12.so: syscall-template.S,82)

Show the call paths in the application run that allocated the largest number of bytes

Using the min_bytes would show all the paths that allocated the minimum number of bytes.

openss>>expview -vfullstack -m max_bytes

Max_Bytes Call Stack Function (defining location)

Read

Written

```

_start (IOR)
> @ 562 in __libc_start_main (libmonitor.so.0.0.0: main.c,541)
>> @ 258 in __libc_start_main (libc-2.12.so: libc-start.c,96)
>>> @ 517 in monitor_main (libmonitor.so.0.0.0: main.c,492)
>>>> @ 153 in main (IOR: IOR.c,108)
>>>>> @ 2013 in TestIoSys (IOR: IOR.c,1848)
>>>>>> @ 2608 in WriteOrRead (IOR: IOR.c,2562)
>>>>>>> @ 244 in IOR_Xfer_POSIX (IOR: aiori-POSIX.c,224)
>>>>>>>> @ 321 in write (iot-collector-monitor-mrnet-mpi.so: wrappers.c,239)
262144 >>>>>>>> @ 82 in write (libc-2.12.so: syscall-template.S,82)
_start (IOR)
> @ 562 in __libc_start_main (libmonitor.so.0.0.0: main.c,541)
>> @ 258 in __libc_start_main (libc-2.12.so: libc-start.c,96)
>>> @ 517 in monitor_main (libmonitor.so.0.0.0: main.c,492)
>>>> @ 153 in main (IOR: IOR.c,108)
>>>>> @ 2173 in TestIoSys (IOR: IOR.c,1848)
>>>>>> @ 2611 in WriteOrRead (IOR: IOR.c,2562)
>>>>>>> @ 251 in IOR_Xfer_POSIX (IOR: aiori-POSIX.c,224)
>>>>>>>> @ 223 in read (iot-collector-monitor-mrnet-mpi.so: wrappers.c,137)
262144 >>>>>>>> @ 82 in __GI__read (libc-2.12.so: syscall-template.S,82)
...

```

Show the top time related call paths in the application run .

openss>>expview -v fullstack

I/O Call Time(ms)	% of Total Time	Number of Calls	Call Stack Function (defining location)
			_start (IOR)
			> @ 562 in __libc_start_main (libmonitor.so.0.0.0: main.c,541)
			>> @ 258 in __libc_start_main (libc-2.12.so: libc-start.c,96)
			>>> @ 517 in monitor_main (libmonitor.so.0.0.0: main.c,492)
			>>>> @ 153 in main (IOR: IOR.c,108)
			>>>>> @ 2021 in TestIoSys (IOR: IOR.c,1848)
			>>>>>> @ 316 in IOR_Close_POSIX (IOR: aiori-POSIX.c,315)
			>>>>>>> @ 766 in close (iot-collector-monitor-mrnet-mpi.so: wrappers.c,685)
1858418.863034	61.486298	512	>>>>>>>> @ 82 in close (libc-2.12.so: syscall-template.S,82)
			_start (IOR)
			> @ 562 in __libc_start_main (libmonitor.so.0.0.0: main.c,541)
			>> @ 258 in __libc_start_main (libc-2.12.so: libc-start.c,96)
			>>> @ 517 in monitor_main (libmonitor.so.0.0.0: main.c,492)
			>>>> @ 153 in main (IOR: IOR.c,108)
			>>>>> @ 2173 in TestIoSys (IOR: IOR.c,1848)
			>>>>>> @ 2611 in WriteOrRead (IOR: IOR.c,2562)
			>>>>>>> @ 251 in IOR_Xfer_POSIX (IOR: aiori-POSIX.c,224)
			>>>>>>>> @ 223 in read (iot-collector-monitor-mrnet-mpi.so: wrappers.c,137)
1055603.730633	34.924939	2048	>>>>>>>>> @ 82 in __GI__read (libc-2.12.so: syscall-template.S,82)
			_start (IOR)
			> @ 562 in __libc_start_main (libmonitor.so.0.0.0: main.c,541)
			>> @ 258 in __libc_start_main (libc-2.12.so: libc-start.c,96)
			>>> @ 517 in monitor_main (libmonitor.so.0.0.0: main.c,492)
			>>>> @ 153 in main (IOR: IOR.c,108)
			>>>>> @ 2004 in TestIoSys (IOR: IOR.c,1848)
			>>>>>> @ 104 in IOR_Create_POSIX (IOR: aiori-POSIX.c,74)
			>>>>>>> @ 670 in open64 (iot-collector-monitor-mrnet-mpi.so: wrappers.c,608)
103350.518692	3.419380	512	>>>>>>>>> @ 82 in __libc_open (libc-2.12.so: syscall-template.S,82)
			_start (IOR)
			> @ 562 in __libc_start_main (libmonitor.so.0.0.0: main.c,541)
			>> @ 258 in __libc_start_main (libc-2.12.so: libc-start.c,96)
			>>> @ 517 in monitor_main (libmonitor.so.0.0.0: main.c,492)
			>>>> @ 153 in main (IOR: IOR.c,108)


```

>>>>> @ 2161 in TestIoSys (IOR: IOR.c,1848)
>>>>>> @ 195 in IOR_Open_POSIX (IOR: aiori-POSIX.c,173)
>>>>>>> @ 670 in open64 (iot-collector-monitor-mrnet-mpi.so: wrappers.c,608)
4757.147988 0.157392 512 >>>>>>>> @ 82 in __libc_open (libc-2.12.so: syscall-template.S,82)
_start (IOR)
> @ 562 in __libc_start_main (libmonitor.so.0.0.0: main.c,541)
>> @ 258 in __libc_start_main (libc-2.12.so: libc-start.c,96)
>>> @ 517 in monitor_main (libmonitor.so.0.0.0: main.c,492)
>>>> @ 153 in main (IOR: IOR.c,108)
>>>>> @ 2013 in TestIoSys (IOR: IOR.c,1848)
>>>>>> @ 2608 in WriteOrRead (IOR: IOR.c,2562)
>>>>>>> @ 244 in IOR_Xfer_POSIX (IOR: aiori-POSIX.c,224)
>>>>>>>> @ 321 in write (iot-collector-monitor-mrnet-mpi.so: wrappers.c,239)
316.176763 0.010461 2048 >>>>>>>>>> @ 82 in write (libc-2.12.so: syscall-template.S,82)

```

6.4 I/O Lightweight Profiling (iop) General Usage

The OSS iop I/O function profiling experiment wraps the most common I/O functions, records the time spent in each I/O function, records the call path along which the I/O function was called, records the time spent along each call path to an I/O function, and records the number of times each function was called.

6.4.1 I/O Profiling (iop) experiment performance data gathering

The I/O Profiling (iop) experiment convenience script is “ossiop”. Here’s how to use this convenience script to gather lightweight I/O profiling performance data: ossiop “how you normally run your application”

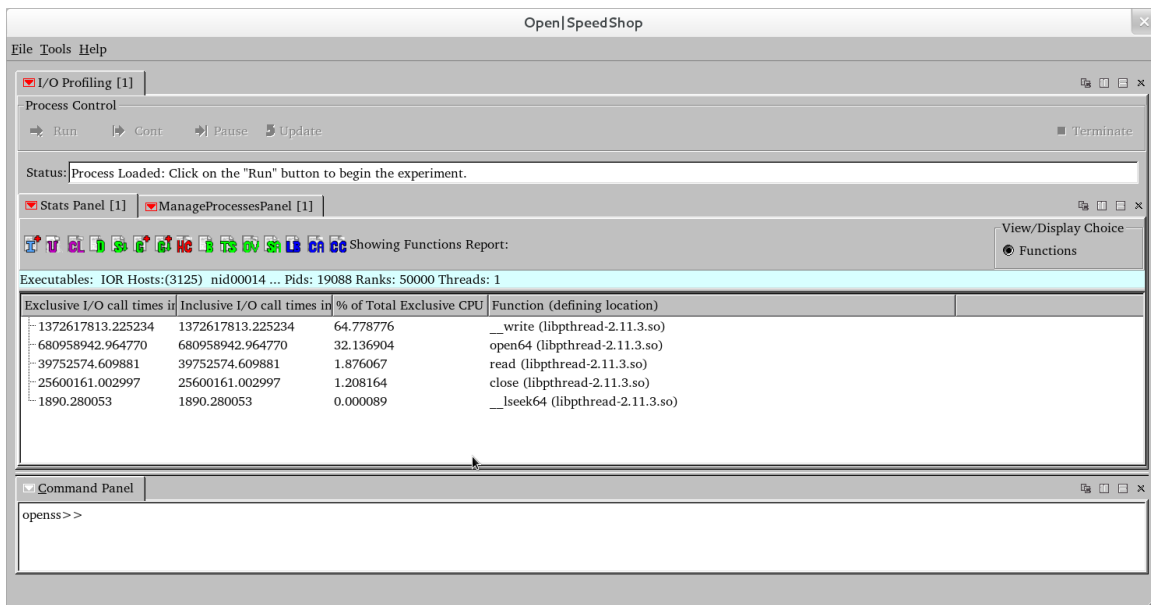
Here’s an example of how to use the ossiop convenience script to gather data for the IOP application on the Cray platform:

```
ossiop "aprun -n 64 ./IOR"
```

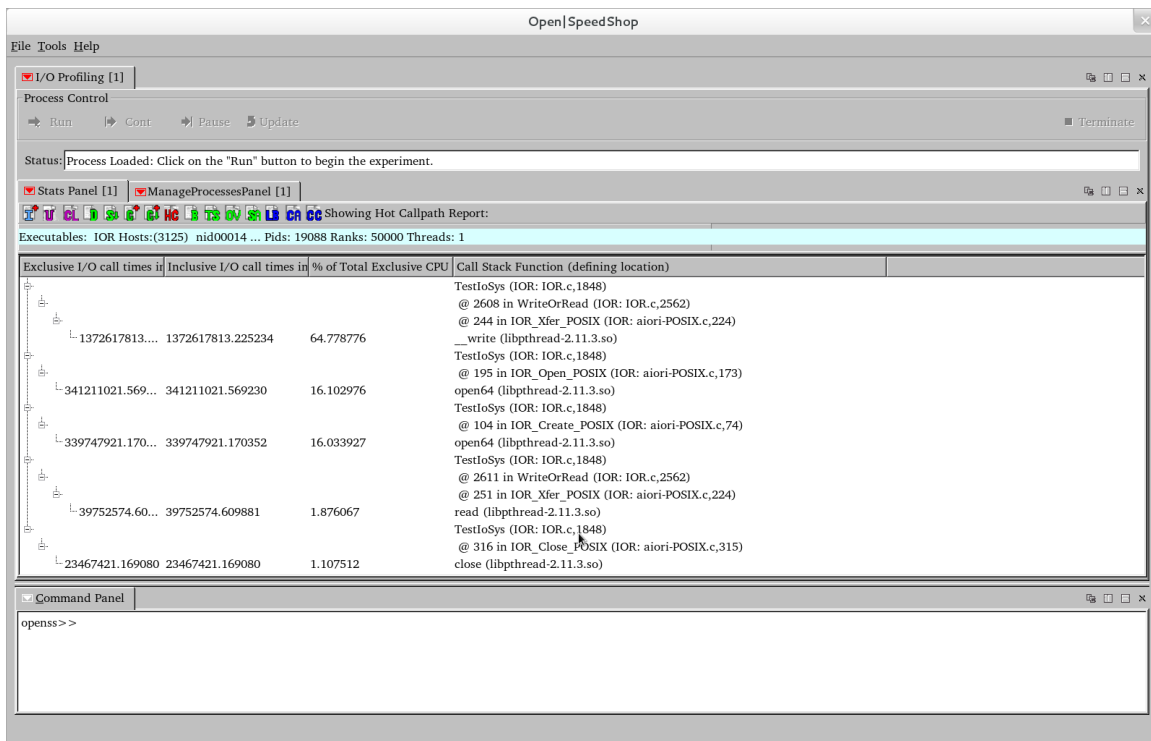
6.4.2 Viewing I/O Profiling (iop) experiment performance data via GUI

To launch the GUI on any experiment, use “openss -f <database name>”.

The image below shows the default view for the iop experiment run on a 50000 rank “IOR” application job. Performance information shown in the default view is the time spent in I/O functions and the percentage of time spent in each I/O function.

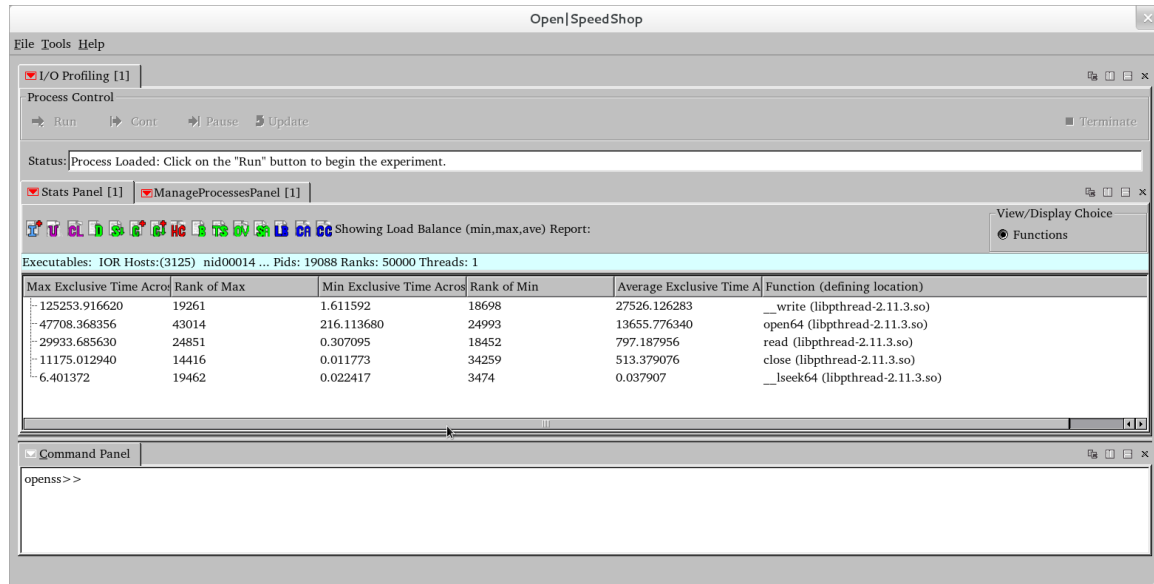


The image below shows the hot call path view for the iop experiment run on a 50000 rank “IOR” application job. The performance information displayed in this view includes the top five call paths to each of the I/O functions that took the most time, time spent in I/O functions and the percentage of time spent in each I/O function.



This image shows the min, max and average time spent in each of the I/O functions and the rank of the minimum value and the rank of the maximum value for each I/O

function. This view indicates whether there is an imbalance relative to the I/O in the application being run. This may or may not be expected.



6.4.3 Viewing I/O Profiling (iop) experiment performance data via CLI

To launch the CLI on any experiment, use “opnss -cli -f <database name>”.

The CLI can provide the same data options as the GUI views. Here are some examples of performance data users can view and the commands to generate the CLI views.

```
> opnss -cli -f IOR-iop-1.opnss
opnss>>[opnss]: The restored experiment identifier is: -x 1
opnss>>expview

Exclusive Inclusive % of Function (defining location)
I/O call I/O call Total
times in times in Exclusive
seconds. seconds. CPU Time
38297.33 38297.33 96.46 __write (libpthread-2.11.3.so)
741.01 741.01 1.86 open64 (libpthread-2.11.3.so)
598.43 598.43 1.50 read (libpthread-2.11.3.so)
63.38 63.38 0.15 close (libpthread-2.11.3.so)
2.264 2.26 0.01 __lseek64 (libpthread-2.11.3.so)

opnss>>expview -v fullstack

Exclusive Inclusive % of Call Stack Function (defining location)
I/O call I/O call Total
times in times in Exclusive
seconds. seconds. CPU Time
TestIoSys (IOR: IOR.c,1848)
> @ 2608 in WriteOrRead (IOR: IOR.c,2562)
>> @ 244 in IOR_Xfer_POSIX (IOR: aiori-POSIX.c,224)
38297.33 38297.33 96.46 >>>__write (libpthread-2.11.3.so)
```

```

TestIoSys (IOR: IOR.c,1848)
> @ 2611 in WriteOrRead (IOR: IOR.c,2562)
>> @ 251 in IOR_Xfer_POSIX (IOR: aiori-POSIX.c,224)
598.43  598.43  1.51 >>>read (libpthread-2.11.3.so)
TestIoSys (IOR: IOR.c,1848)
> @ 104 in IOR_Create_POSIX (IOR: aiori-POSIX.c,74)
472.14  472.14  1.19 >>open64 (libpthread-2.11.3.so)
TestIoSys (IOR: IOR.c,1848)
> @ 195 in IOR_Open_POSIX (IOR: aiori-POSIX.c,173)
268.88  268.88  0.68 >>open64 (libpthread-2.11.3.so)
TestIoSys (IOR: IOR.c,1848)
> @ 316 in IOR_Close_POSIX (IOR: aiori-POSIX.c,315)
61.587482  61.587482  0.155123 >>close (libpthread-2.11.3.so)
TestIoSys (IOR: IOR.c,1848)
> @ 316 in IOR_Close_POSIX (IOR: aiori-POSIX.c,315)
1.796442  1.796442  0.004525 >>close (libpthread-2.11.3.so)
TestIoSys (IOR: IOR.c,1848)
> @ 2608 in WriteOrRead (IOR: IOR.c,2562)
>> @ 234 in IOR_Xfer_POSIX (IOR: aiori-POSIX.c,224)
1.280113  1.280113  0.003224 >>>_lseek64 (libpthread-2.11.3.so)
TestIoSys (IOR: IOR.c,1848)
> @ 2611 in WriteOrRead (IOR: IOR.c,2562)
>> @ 234 in IOR_Xfer_POSIX (IOR: aiori-POSIX.c,224)
0.981341  0.981341  0.002472 >>>_lseek64 (libpthread-2.11.3.so)

```

In the CLI output above, the `expview` command with no options gives the overview or summary view for all the ranks and threads. Users can view the performance information for individual ranks (using `-r <rank number>`), individual threads (using `-t <thread number>`) or individual processes (using `-p <process id>`). Users also can give a range of ranks, threads or processes using their respective option.

The calltrees view shows where the I/O function was called from in the user's application source. In this example, most I/O time was spent in the write I/O function along the path shown in the first individual call path. The call path with `fullstack` option stops the calltrees view from collapsing any similar sub-trees, which makes the view more explicit. Without the `fullstack` option the calltrees would be more consolidated.

7 Applying Experiments to Parallel Codes

The ideal scenario for executing parallel code using pthreads or OpenMP is efficient threading, in which all threads are assigned work that can execute concurrently. In the ideal scenario for MPI code, the job is properly load balanced so all MPI ranks do the same amount of work and none is stuck waiting.

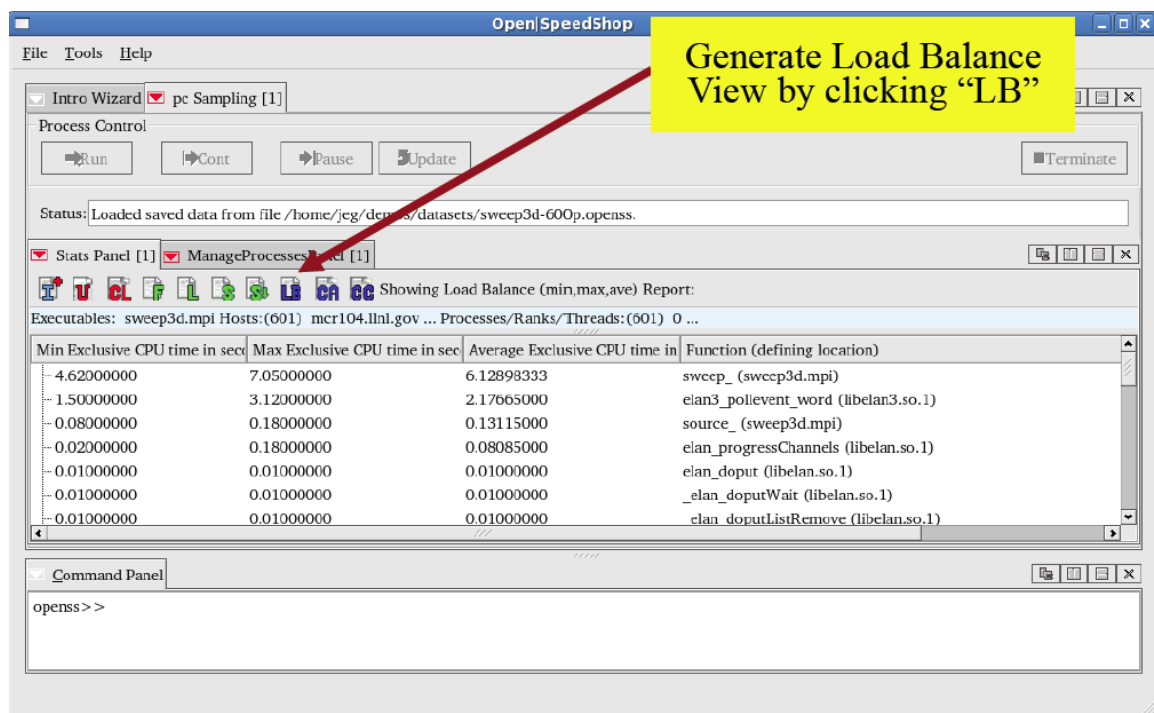
What can make these ideal scenarios fail? According to a Lawrence Livermore National Laboratory parallel processing tutorial, MPI jobs can become unbalanced if an unequal amount of work was assigned to each rank, possibly through an unequal number of array operations for each rank or through uneven distribution of loop iterations. Problems can persist even if the work seems to be evenly distributed. For example, when a sparsely populated array is evenly distributed, some ranks may end up with very little or no work while others will have a full load. Under adaptive grid models, some ranks must redefine their mesh while others don't. Under N-body simulations, some work migrates to other ranks so those ranks will have more to do while the others have less.

Performance analysis can help with load balancing and evenly distributing work. Tools like O|SS are designed to work on parallel jobs. It supports threading and message passing and automatically tracks all ranks and threads during execution. It also can store the performance information on a per process, rank or thread basis for individual evaluation. All of the O|SS experiments run on parallel jobs and collectors are applied to all ranks on all nodes. An experiment's results can be displayed as an aggregation across all ranks or threads (the default view) or the user can select individual or groups of ranks or threads to view. There also are experiments specifically designed for tracing MPI function calls.

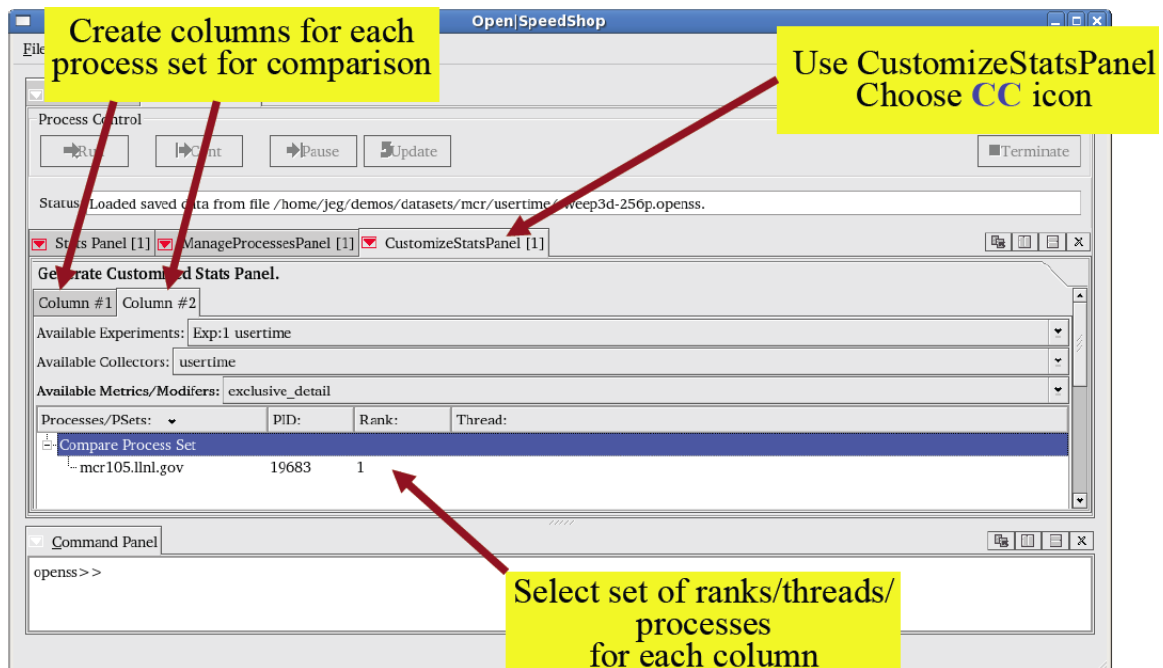
O|SS has been tested with a variety of MPI versions, including Open MPI, MVAPICH[2] and MPICH2 on Intel, Blue Gene and Cray systems. O|SS can identify the MPI task (rank info) through the MPIR interface for the online version or through PMPI preload for the offline version. To run MPI code with O|SS, just include the MPI launcher as part of the executable as normal. Here are several examples:

```
> ossmpi "mpirun -np 128 sweep3d.mpi"
> osspcsamp "mpirun -np 32 sweep3d.mpi"
> ossio "srun -N 4 -n 16 sweep3d.mpi"
> openss -offline -f "mpirun -np 128 sweep3d.mpi" hwctime
> openss -online -f "srun -N 8 -n 128 sweep3d.mpi" usertime
```

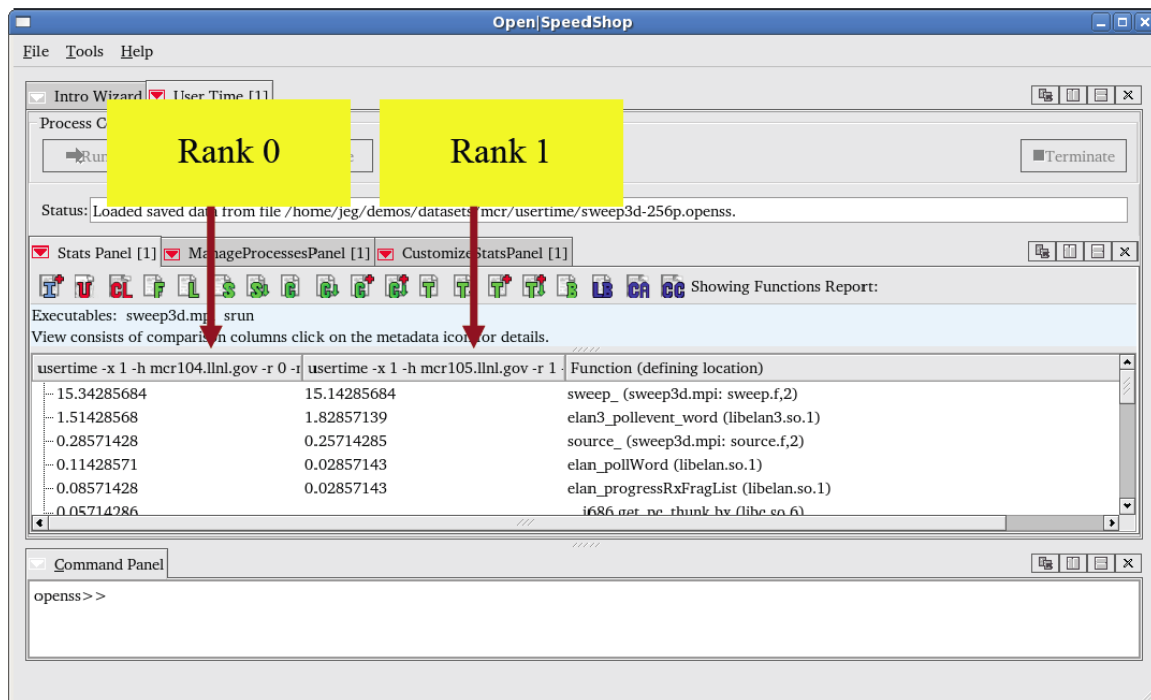
The default view for parallel applications aggregates the information collected across all ranks. Users can manually include or exclude individual ranks, processes or threads to view their specific results. They also can use the Customize Stats Panel View to compare ranks and can create a compare column for the process groups or individual ranks. Cluster analysis also is available and can be used to find outliers – ranks that are performing very differently from others. From the Stats Panel toolbar or context menu users can automatically create groups of similarly performing ranks or threads. Through the Stat Panel, OJSS also provides common analysis functions designed for quick analysis of MPI applications. Load-balance views that calculate min, max and average values across ranks, processes or threads are available. This image shows the OJSS buttons for Load Balance. Cluster Analysis is next to that.



This shows the creation of a comparison between two ranks in OJSS:



This shows those two ranks compared side by side in the statistics panel:



8 MPI Tracing Experiments (mpi, mpit, mpip)

This section follows an O|SS MPI tracing experiment that will record all MPI call invocations. There are three MPI experiments and associated convenience scripts: ossmpi, which records call times; ossmpit, which records call times and arguments;

and mpip, a lightweight version of mpi that records individual MPI calls but doesn't save them in the database. Equal events will be aggregated to save space in the database and reduce the overhead.

Again, we will run the experiment on the SMG2000 application. Syntax for the experiment is:

```
> ossmpi[t][p] "srun -N 4 -n 32 smg2000 -n 50 50 50" [default | <list MPI functions> | mpi_category]
```

The default behavior is to trace all MPI functions, but a comma-separated list can be supplied if users only want to trace specific ones, e.g. MPI_Send, MPI_Recv..., etc. Users also can select an mpi_category to trace "all", "asynchronous_p2p", "collective_com", "datatypes", "environment", "graphs_contexts_comms", "persistent_com", "process_topologies" and "synchronous_p2p".

The default views are designed to relate the information included in the report back to the individual calls to their corresponding MPI functions. This same information would be reported if the user were to do an: "expview -m min, max, average". The view is a representation of the minimum, maximum and average time values per individual calls to their corresponding MPI functions.

The average time reported is the total time for all calls to a function divided by the total number of calls. Thus, it is the average time that each individual call spends in the function. As such, it is comparable to the Max (maximum) and Min (minimum) of a call to the function found in the same "min, max, average" report.

Alternatively, if a user does an "expview -m ThreadMin, ThreadMax, ThreadAve", then information for the Max, Min and Average is related back to the individual ranks.

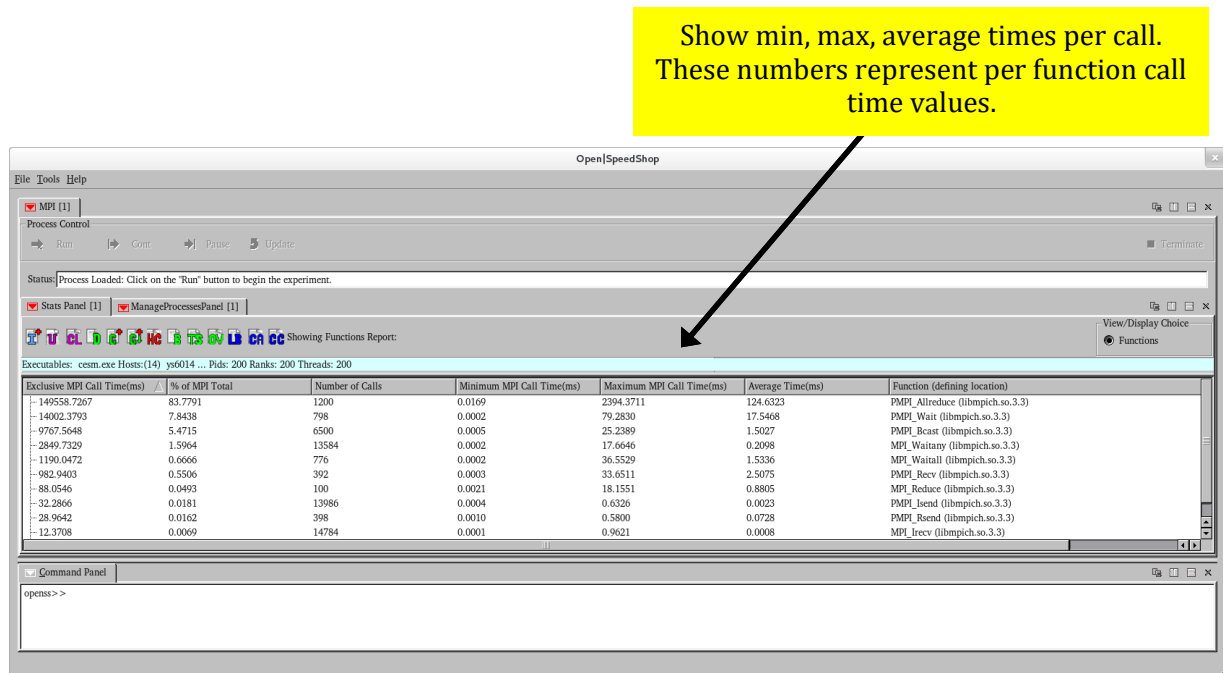
Another way of saying it: The average is the total amount of time for all calls to a function divided by the total number of ranks. Thus, it is the average time each rank spends in the function. As such, it is comparable to the Max and Min of a rank in the same report.

If the number of ranks is the same as the number of calls, the two different calculations should produce the same result. This would be true if all calls were in a single thread or there were one in each rank, as it is for MPI_Init.

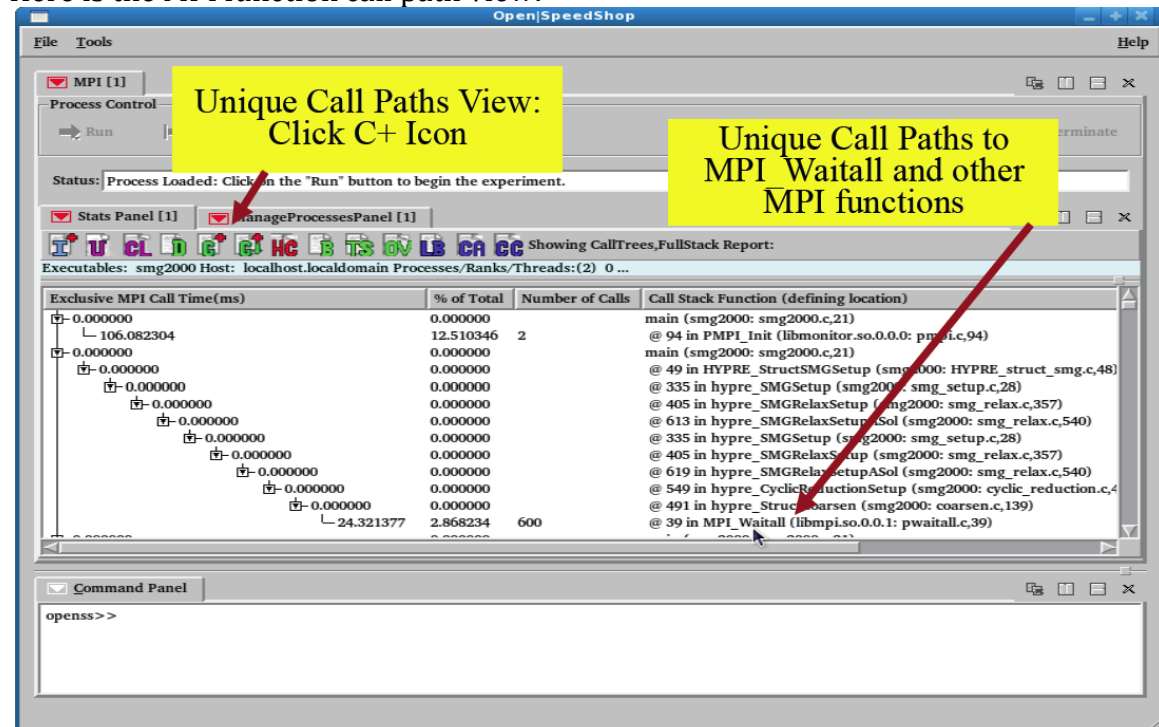
The "expview -m min, max, average" view can expose load imbalance by showing large differences in the minimum and maximum times for asynchronous MPI functions. This indicates that some of the MPI asynchronous functions ran quickly (low minimum times) but some had to wait a long time to get started (large maximum times). Many times, the function calls that ran quickly were the last to arrive and actually are from ranks that are running worse than the others, causing load imbalance and delaying the overall job execution. These ranks show better MPI

function time numbers, but only because they were the last to arrive at the internal barrier point and did not have to wait as long as other MPI functions that arrived sooner, but had to wait for the other ranks to arrive.

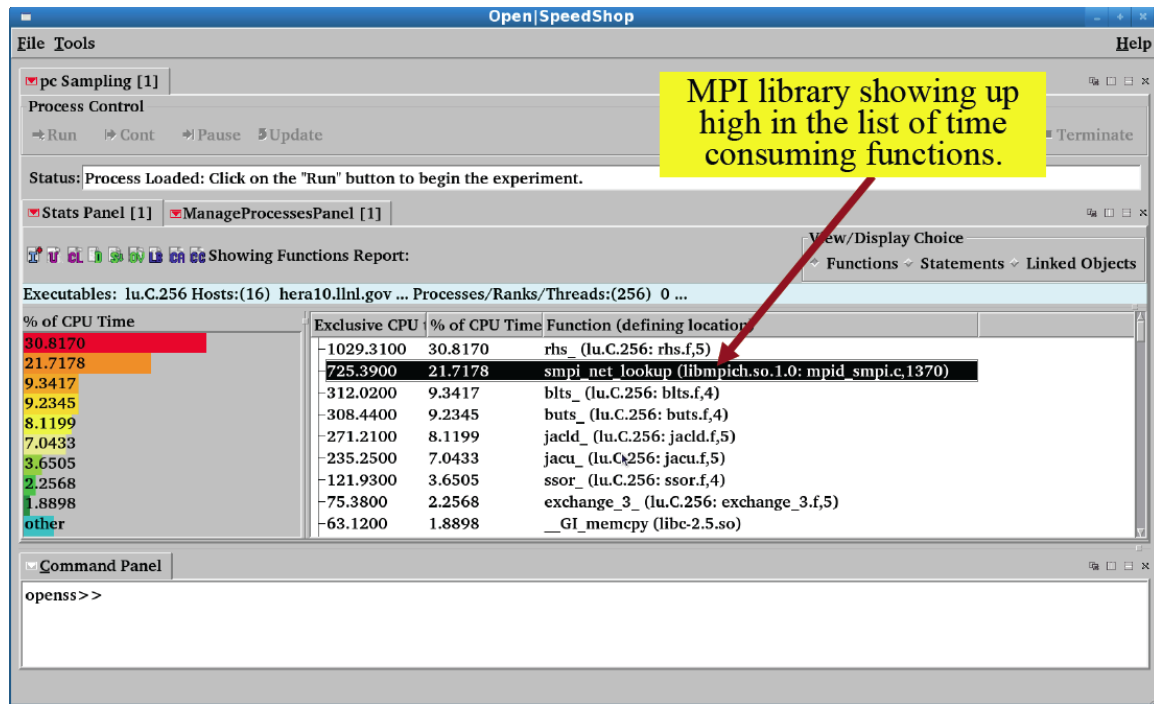
This shows results of the MPI experiment in the default view:



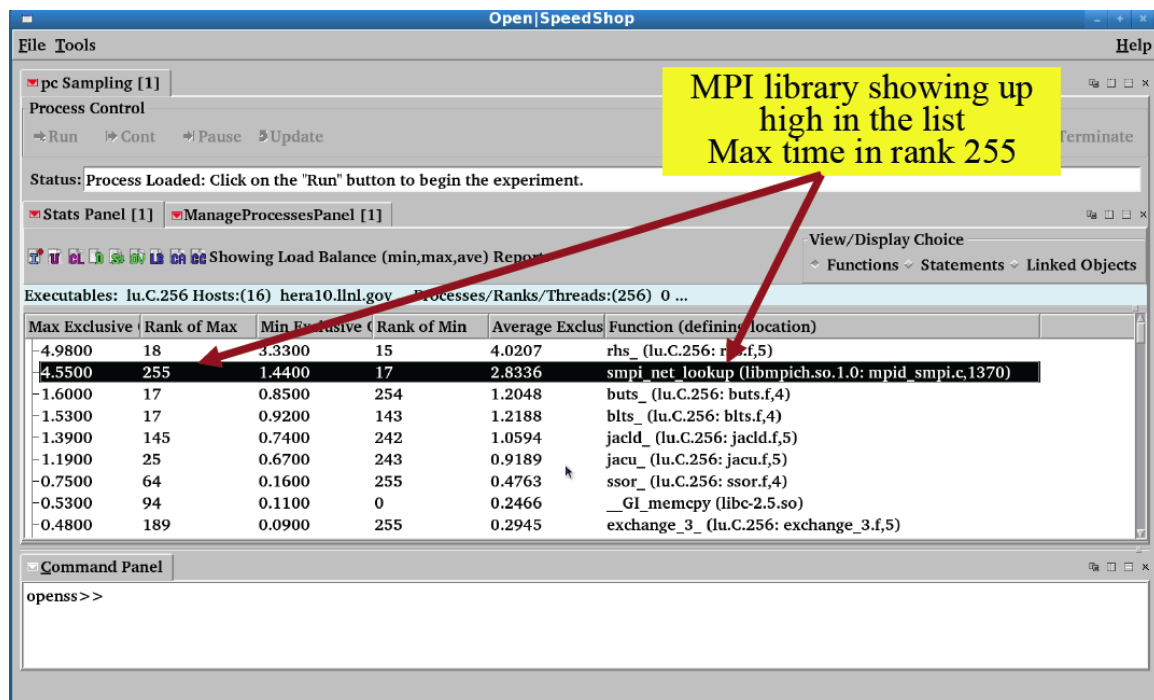
Here is the MPI function call path view:



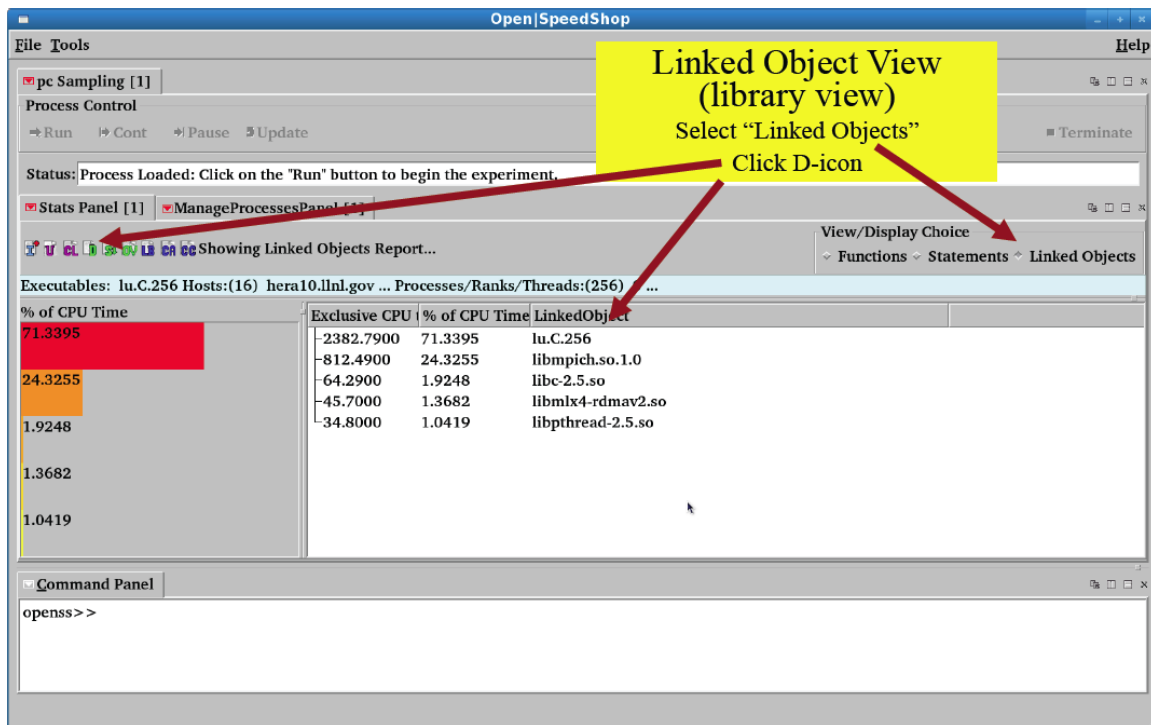
Here is the default pcsamp view based on functions:



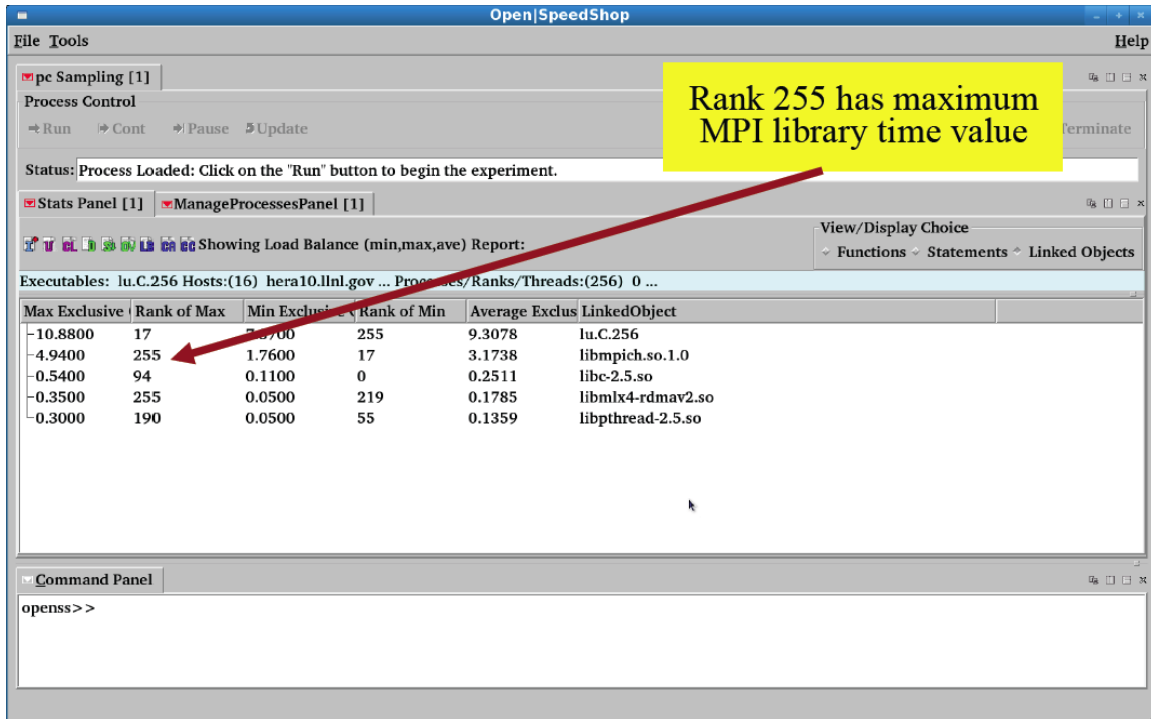
Here is the load balance view based on functions:



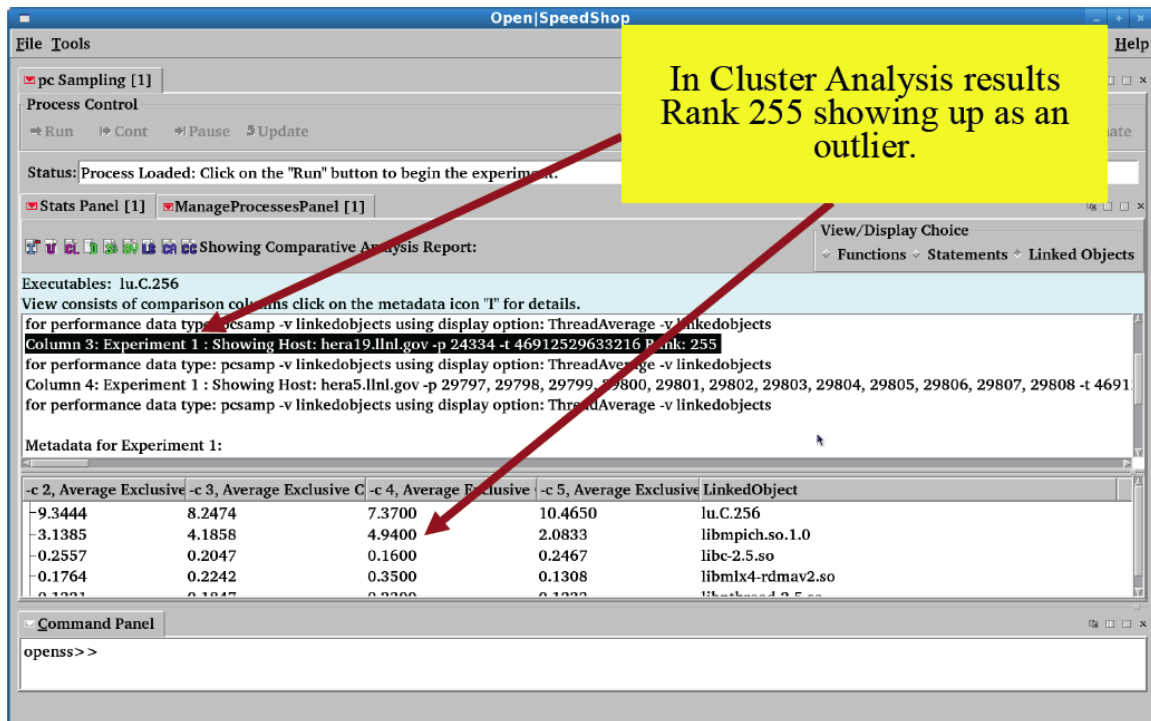
Here is the default view based on Linked Objects (libraries):



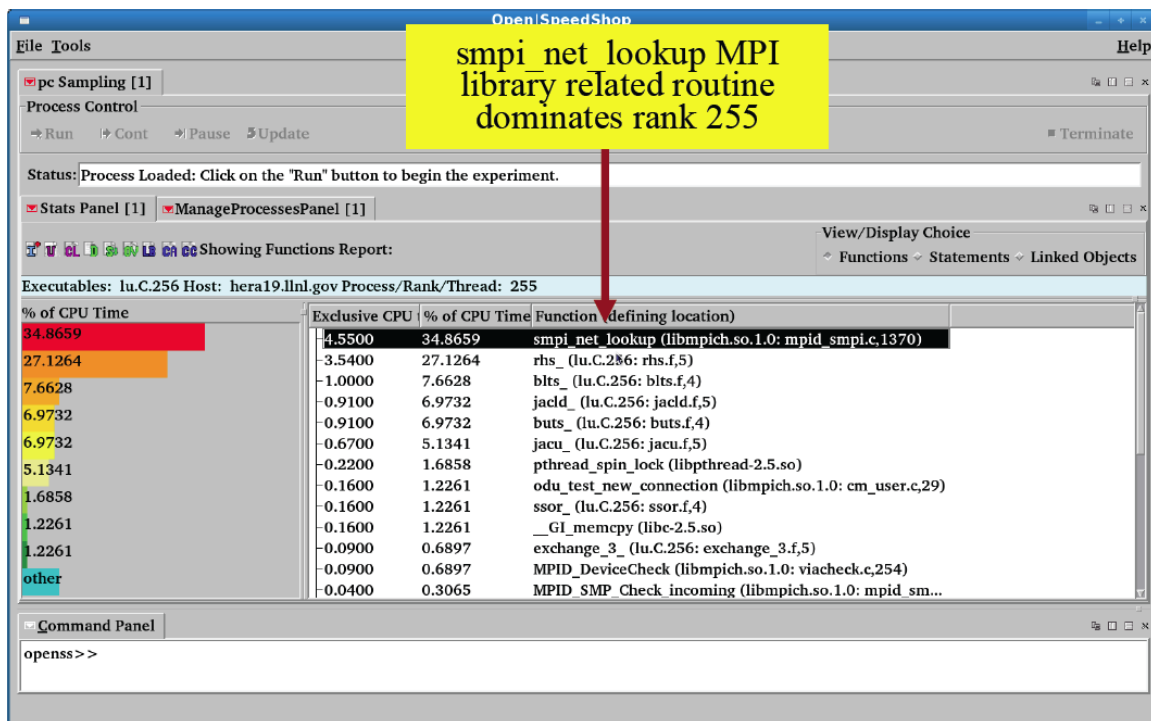
This is the load balance view base on Linked Objects (libraries):



Here is the cluster analysis view based on Linked Objects:



Here is the pcsamp view of Rank 255 performance data only:



Rank 255 is examined further here, this time using the load balance view in the Command Line Interface for OJSS:

openss>>expview -m loadbalance

Max MPI Call Time (defining location)	Rank of Max	Min MPI Call Time	Rank of Min	Average MPI Call Function
Across Ranks(ms)		Across Ranks(ms)		Time Across Ranks(ms)
150332.97	0	120351.97	36	131361.13 MPI_Recv (libmpich.so.1.0: recv.c,60)
17636.11	36	1103.53	0	5443.08 MPI_Send (libmpich.so.1.0: send.c,65)
16470.53	19	353.81	0	5255.33 MPI_Wait (libmpich.so.1.0: wait.c,51)
3206.45	255	3.00	17	2000.27 MPI_Allreduce (libmpich.so.1.0: allreduce.c,59)
915.17	54	754.39	83	792.07 PMPI_Init (libmonitor.so.0.0.0: pmpi.c,94)
16.00	48	5.63	249	7.29 PMPI_Finalize (libmonitor.so.0.0.0: pmpi.c,223)
9.28	230	2.99	0	7.55 MPI_Irecv (libmpich.so.1.0: irecv.c,48)
1.22	247	0.07	0	1.10 MPI_Bcast (libmpich.so.1.0: bcast.c,81)
0.51	53	0.35	239	0.41 MPI_Barrier (libmpich.so.1.0: barrier.c,56)

openss>>

MPI Experiment shows Rank 255 spending significant time in MPI_Allreduce

This shows the difference between Rank 255 and Rank 0:

openss>>expview -r 255 -m exclusive_time

openss>>expview -r 0 -m exclusive_time

Exclusive MPI Call Function (defining location)	Exclusive MPI Call Function (defining location)
Time(ms)	Time(ms)
138790.370000 MPI_Recv (libmpich.so.1.0: recv.c,60)	150332.974000 MPI_Recv (libmpich.so.1.0: recv.c,60)
8841.088000 MPI_Wait (libmpich.so.1.0: wait.c,51)	1103.539000 MPI_Send (libmpich.so.1.0: send.c,65)
3337.737000 MPI_Send (libmpich.so.1.0: send.c,65)	807.433000 PMPI_Init (libmonitor.so.0.0.0: pmpi.c,94)
3206.454000 MPI_Allreduce (libmpich.so.1.0: allreduce.c,59)	353.810000 MPI_Wait (libmpich.so.1.0: wait.c,51)
797.964000 PMPI_Init (libmonitor.so.0.0.0: pmpi.c,94)	12.643000 PMPI_Finalize (libmonitor.so.0.0.0: pmpi.c,223)
5.887000 PMPI_Finalize (libmonitor.so.0.0.0: pmpi.c,223)	8.903000 MPI_Allreduce (libmpich.so.1.0: allreduce.c,59)
4.701000 MPI_Irecv (libmpich.so.1.0: irecv.c,48)	2.995000 MPI_Irecv (libmpich.so.1.0: irecv.c,48)
1.221000 MPI_Bcast (libmpich.so.1.0: bcast.c,81)	0.438000 MPI_Barrier (libmpich.so.1.0: barrier.c,56)
0.396000 MPI_Barrier (libmpich.so.1.0: barrier.c,56)	0.076000 MPI_Bcast (libmpich.so.1.0: bcast.c,81)

MPI Experiment comparison of rank 255 to another (rank 0) shows Rank 255 spending much more time in MPI_Allreduce and MPI_Wait

Here are the hot call paths for MPI_Wait on Rank 255:

openss>>expview -r 255 -vcalltrees,fullstack -f MPI_Wait

Exclusive MPI Call Time(ms)	% of Total	Number of Calls	Call Stack Function (defining location)
		>>>>main (lu.C.256)	
		>>>> @ 140 in MAIN__ (lu.C.256: lu.f,46)	
		>>>>> @ 180 in ssor_ (lu.C.256: ssor.f,4)	
		>>>>>> @ 213 in rhs_ (lu.C.256: rhs.f,5)	
		>>>>>>> @ 224 in exchange_3_ (lu.C.256: exchange_3.f,5)	
		>>>>>>>> @ 893 in mpi_wait_ (mpi-mvapich-rt-offline.so: wrappers-fortran.c,893)	
6010.978000	3.878405	>>>>>>>> @ 889 in mpi_wait (mpi-mvapich-rt-offline.so: wrappers-fortran.c,885)	
		250 >>>>>>>>> @ 51 in MPI_Wait (libmpich.so.1.0: wait.c,51)	
		>>>>main (lu.C.256)	
		>>>> @ 140 in MAIN__ (lu.C.256: lu.f,46)	
		>>>>> @ 180 in ssor_ (lu.C.256: ssor.f,4)	
		>>>>>> @ 64 in rhs_ (lu.C.256: rhs.f,5)	
		>>>>>>> @ 88 in exchange_3_ (lu.C.256: exchange_3.f,5)	
		>>>>>>>> @ 893 in mpi_wait_ (mpi-mvapich-rt-offline.so: wrappers-fortran.c,893)	
2798.770000	1.805823	>>>>>>>> @ 889 in mpi_wait (mpi-mvapich-rt-offline.so: wrappers-fortran.c,885)	
		250 >>>>>>>>> @ 51 in MPI_Wait (libmpich.so.1.0: wait.c,51)	

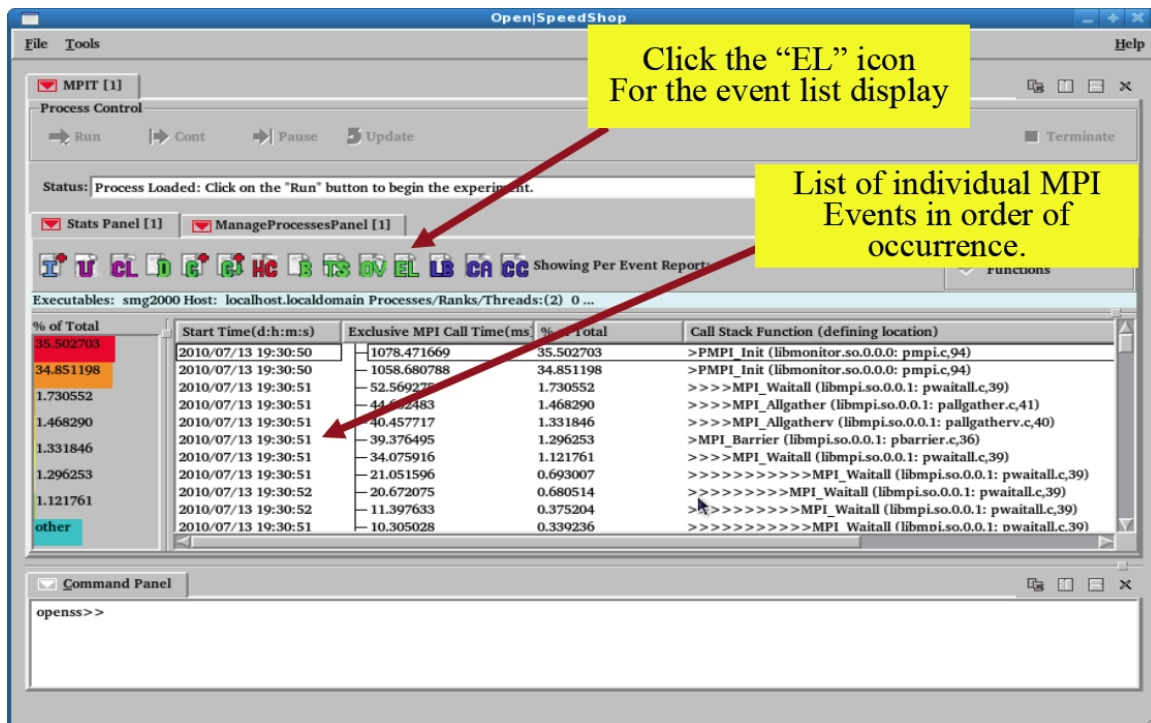
Most expensive call path to MPI_Wait

In this experiment, we did program counter sampling to get an overview of the application. We noticed that `smp_net_lookup` appeared in function load balance view, prompting an examination of the linked object view. The load balance on the linked object showed some imbalance, so we looked at the cluster analysis view and found that rank 255 was an outlier.

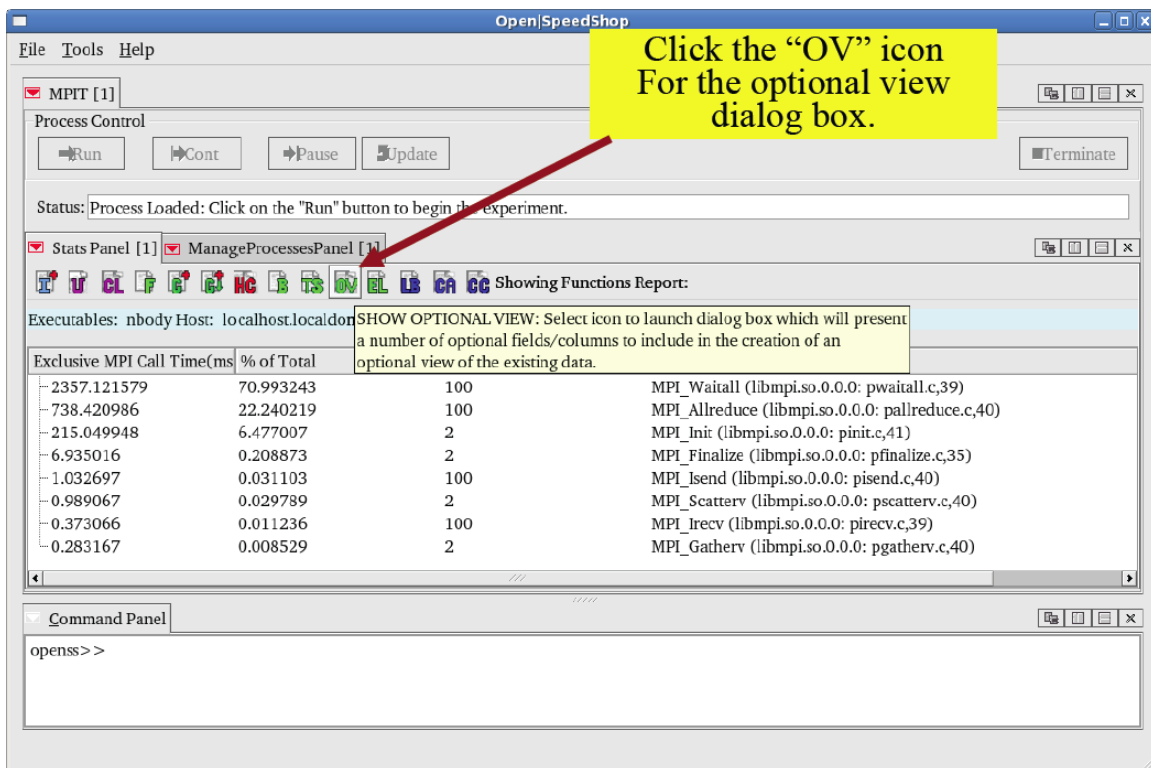
Taking a closer look at rank 255, we saw that the pcsamp output shows most of the time was spent in `smp_net_lookup`. To get more clues, we used the MPI experiment and saw that a load balance view shows rank 255's `MPI_Allreduce` time is the highest of the 256 ranks. We then checked rank 255 and a representative rank from the rest and noted the differences in `MPI_Wait`, `MPI_Send` and `MPI_Allreduce`. We looked at the call paths to `MPI_Wait` to determine why the wait was occurring.

The mpit experiment has a performance information entry for each MPI function call. Besides time spent in each MPI function, information such as source and destination rank and bytes sent or received also are available. Users can selectively view the information they desire.

Here is the default event view for an MPI application:



User can create their own event view with the OV button:



Use the views dialog box to choose the metrics to display:

8.1.1 MPI Tracing Experiment (mpi) performance data gathering

Much of this information is described in the main MPI Tracing Experiments section (above), but for completeness, here is the convenience script description for running the MPI-specific tracing experiments:

```
> ossmpi "srun -N 4 -n 32 smg2000 -n 50 50 50" [default | <list MPI functions> | mpi_category]
```

If users give the mpi_category or a list of categories to the ossmpi command, then only those MPI functions corresponding to that category or categories will be traced. This table defines the MPI categories:

MPI Category	Argument
All MPI Functions	all
Collective Communicators	collective_com
Persistent Communicators	persistent_com
Synchronous Point to Point	synchronous_p2p
Asynchronous Point to Point	asynchronous_p2p
Process Topologies ^[1] _{SEP}	process_topologies
Groups Contexts Communicators	graphs_contexts_comms
Environment ^[1] _{SEP}	environment ^[1] _{SEP}
Datatypes	datatypes
MPI File I/O	fileio

8.1.2 Viewing MPI Tracing Experiment (mpi) performance data via GUI

To launch the GUI on any experiment, use “openss -f <database name>”.

8.1.3 MPI Viewing Tracing Experiment (mpi) performance data via CLI

To launch the CLI on any experiment, use “openss -cli -f <database name>”. This table describes the header and column data definitions for the default MPI experiment views.

Column Name	Column Definition
Exclusive MPI Call Time	Aggregated total exclusive time spent in the MPI function corresponding to this row of data.
% of MPI Time	Percentage of exclusive MPI time spent in the MPI function corresponding to this row of data relative to the total MPI time for all the MPI functions.
Number of Calls	Total number of calls to the MPI function corresponding to this row of data.
Min MPI Call Time	The minimum time that an MPI call took across all calls spent in the corresponding MPI function.
Max MPI Call Time	The maximum time that an MPI call took across all calls spent in the corresponding MPI function.

Column Name	Column Definition
Average MPI Call Time Across Ranks	The average time for the default view is the total amount of time for all the calls to a function divided by the total number of calls. Thus, it is the average time that each MPI function call spends in the function.

This is an example of the CLI default view for the MPI (mpi, mpit) experiments:

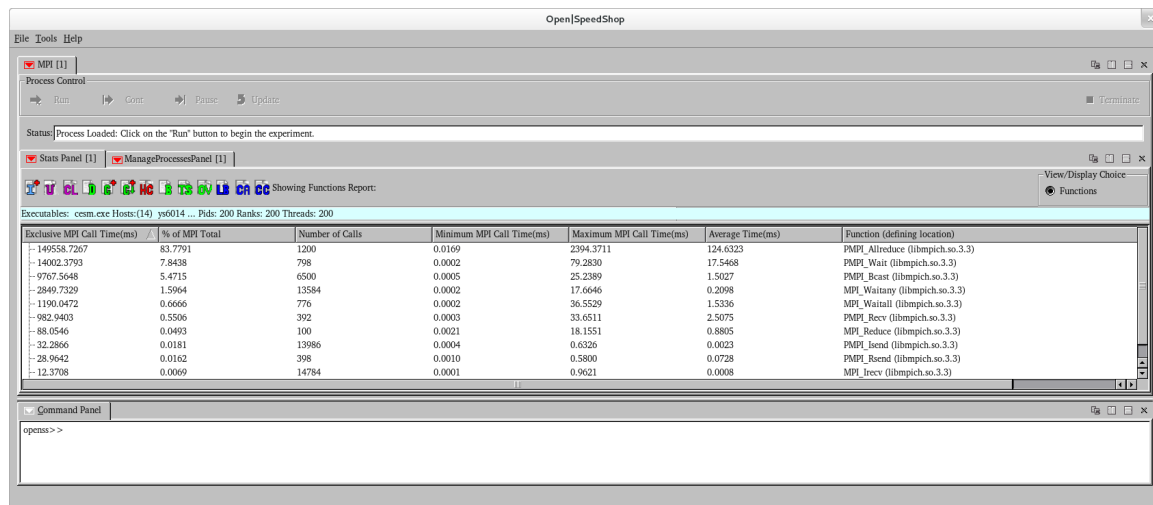
```
jeg@localhost:~/ucar
File Edit View Search Terminal Help
openss>>[openss]: The restored experiment identifier is: -x 1
openss>>expview

Exclusive MPI Call Time(ms)    % of MPI Total    Number of Calls    Minimum MPI Call Time(ms)    Maximum MPI Call Time(ms)    Average Time(ms)    Function (defining location)

149558.7267    83.7791    1200    0.0169    2394.3711    124.6323    PMPI_Allreduce (libmpich.so.3.3)
14002.3793    7.8438    798    0.0002    79.2830    17.5468    PMPI_Wait (libmpich.so.3.3)
9767.5648    5.4715    6500    0.0005    25.2389    1.5027    PMPI_Bcast (libmpich.so.3.3)
2849.7329    1.5964    13584    0.0002    17.6646    0.2098    MPI_Waitany (libmpich.so.3.3)
1190.0472    0.6666    776    0.0002    36.5529    1.5336    MPI_Waitall (libmpich.so.3.3)
982.9403    0.5506    392    0.0003    33.6511    2.5075    PMPI_Recv (libmpich.so.3.3)
88.0546    0.0493    100    0.0021    18.1551    0.8805    MPI_Reduce (libmpich.so.3.3)
32.2866    0.0181    13986    0.0004    0.6326    0.0023    PMPI_Isend (libmpich.so.3.3)
28.9642    0.0162    398    0.0010    0.5800    0.0728    PMPI_Rsend (libmpich.so.3.3)
12.3708    0.0069    14784    0.0001    0.9621    0.0008    MPI_Irecv (libmpich.so.3.3)
2.4564    0.0014    792    0.0006    0.0169    0.0031    PMPI_Send (libmpich.so.3.3)

openss>>
```

This is an example of the GUI default view for the MPI (mpi, mpit) experiments:



The default views are designed to relate information included in the report back to the individual calls to their corresponding MPI functions. This same information would be reported by using the command: “expview -m min, max, average”. The view is a representation of the minimum, maximum and average time values per individual calls to their corresponding MPI functions.

The average time reported is the total time for all calls to a function divided by the total number of calls. Thus, it is the average time each individual call spends in the

function. As such, it is comparable to the Max (maximum) and Min (minimum) of a call to the function found in the same “min, max, average” report.

Alternatively, if a user does an “expview -m ThreadMin, ThreadMax, ThreadAve”, then the report information for the Max, Min and Average is related back to the individual ranks.

Another way of saying it: The average is the total amount of time for all the calls to a function divided by the total number of ranks. Thus, it is the average time that each rank spends in the function. As such, it is comparable to the Max and Min of a rank in the same report.

If the number of ranks is the same as the number of calls, the two different calculations should produce the same result. This would be true if all the calls were in a single thread or there were one in each rank, as it is for MPI_Init.

The “expview -m min, max, average” view can expose load imbalance by showing when the minimum and maximum time for asynchronous MPI functions have large differences. This indicates that some of the MPI asynchronous functions ran quickly (low minimum times) but some had long waits to get started (large maximum times). Many times, the function calls that ran quickly were the last to arrive and actually are from ranks that are not running as well as the others, causing load imbalance and slowing overall job execution. These ranks show better MPI function time performance, but only because they were the last to arrive at the internal barrier point. They did not have to wait as long as other MPI functions that arrived earlier, but had to wait for the other ranks to finally arrive.

8.2 MPI Tracing Experiments (mpit)

8.2.1 MPI Tracing Experiments (mpit) performance data gathering

Much of this information is described in the main MPI Tracing Experiments section (above), but for completeness, here is the convenience script description for running the MPI-specific tracing experiments:

```
> ossmpit “srun -N 4 -n 32 smg2000 -n 50 50 50” [default | <list MPI functions> | mpi_category]
```

If users give the mpi_category or a list of categories to the ossmpit command, then only those MPI functions corresponding to that category or categories will be traced. This table defines the MPI categories:

MPI Category	Argument
All MPI Functions	all
Collective Communicators	collective_com
Persistent Communicators	persistent_com
Synchronous Point to Point	synchronous_p2p
Asynchronous Point to Point	asynchronous_p2p
Process Topologies	process_topologies

Groups Contexts Communicators Environment ^[1] _{SEP} Datatypes MPI File I/O	graphs_contexts_comms environment ^[1] _{SEP} datatypes fileio
---	---

8.2.2 Viewing MPI Tracing Experiments (mpit) performance data via GUI

To launch the GUI on any experiment, use “openss -f <database name>”.

8.2.3 Viewing MPI Tracing Experiments (mpit) performance data via CLI

To launch the CLI on any experiment, use “openss -cli -f <database name>”.
This table describes the header and column data definitions for the default MPI experiment views.

Column Name	Column Definition
Exclusive MPI Call Time	Aggregated total exclusive time spent in the MPI function corresponding to this row of data.
% of MPI Time	Percentage of exclusive MPI time spent in the MPI function corresponding to this row of data relative to the total MPI time for all the MPI functions.
Number of Calls	Total number of calls to the MPI function corresponding to this row of data.
Min MPI Call Time	The minimum time that an MPI call took across all calls spent in the corresponding MPI function.
Max MPI Call Time	The maximum time that an MPI call took across all calls spent in the corresponding MPI function.
Average MPI Call Time Across Ranks	The average time for the default view is the total amount of time for all the calls to a function divided by the total number of calls. Thus, it is the average time that each MPI function call spends in the function.

This is an example of the CLI default view for the MPI (mpi, mpit) experiments:

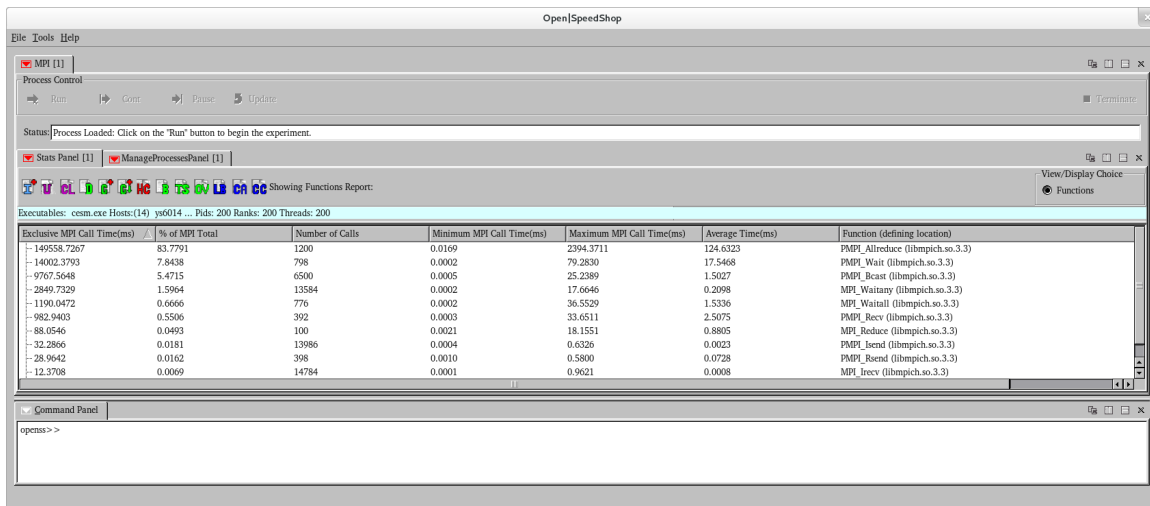
```

jcg@localhost:~/ucar
File Edit View Search Terminal Help
openss>>[openss]: The restored experiment identifier is: -x 1
openss>>expview

Exclusive MPI Call Time(ms)  % of MPI Total  Number of Calls  Minimum MPI Call Time(ms)  Maximum MPI Call Time(ms)  Average Time(ms)  Function (defining location)
149558.7267  83.7791  1200  0.0169  2394.3711  124.6323  PMPI_Allreduce (libmpich.so.3.3)
14002.3793  7.8438  798  0.0002  79.2830  17.5468  PMPI_Wait (libmpich.so.3.3)
9767.5648  5.4715  6500  0.0005  25.2389  1.5027  PMPI_Bcast (libmpich.so.3.3)
2849.7329  1.5964  13584  0.0002  17.6646  0.2098  MPI_Waitany (libmpich.so.3.3)
1190.0472  0.6666  776  0.0002  36.5529  1.5336  MPI_Waitall (libmpich.so.3.3)
982.9403  0.5506  392  0.0003  33.6511  2.5075  PMPI_Recv (libmpich.so.3.3)
88.0546  0.0493  100  0.0021  18.1551  0.8805  MPI_Reduce (libmpich.so.3.3)
32.2866  0.0181  13986  0.0004  0.6326  0.0023  PMPI_Isend (libmpich.so.3.3)
28.9642  0.0162  398  0.0010  0.5800  0.0728  PMPI_Rsend (libmpich.so.3.3)
12.3708  0.0069  14784  0.0001  0.9621  0.0008  MPI_Irecv (libmpich.so.3.3)
2.4564  0.0014  792  0.0006  0.0169  0.0031  PMPI_Send (libmpich.so.3.3)
openss>>

```

This is an example of the GUI default view for the MPI (mpi, mpit) experiments:



The default views are designed to relate information included in the report back to the individual calls to their corresponding MPI functions. This same information that would be reported by doing: “expview -m min, max, average”. The view is a representation of the minimum, maximum and average time values per individual calls to their corresponding MPI functions.

The average time reported is the total time for all calls to a function divided by the total number of calls. Thus, it is the average time each individual call spends in the function. As such, it is comparable to the Max (maximum) and Min (minimum) of a call to the function found in the same “min, max, average” report.

Alternatively, if a user does an “expview -m ThreadMin, ThreadMax, ThreadAve”, then the report information for the Max, Min and Average is related back to the individual ranks.

Another way of saying it: The average is the total amount of time for all the calls to a function divided by the total number of ranks. Thus, it is the average time that each

rank spends in the function. As such, it is comparable to the Max and Min of a rank in the same report.

If the number of ranks is the same as the number of calls, the two different calculations should produce the same result. This would be true if all the calls were in a single thread or there were one in each rank, as it is for MPI_Init.

The “expview -m min, max, average” view can expose load imbalance by showing when the minimum and maximum time for asynchronous MPI functions have large differences. This indicates that some of the MPI asynchronous functions ran quickly (low minimum times) but some had long waits to get started (large maximum times). Many times the function calls that ran quickly were the last to arrive and actually are from ranks that are not running as well as the others, causing load imbalance and slowing overall job execution. These ranks show better MPI function time performance, but only because they were the last to arrive at the internal barrier point. They did not have to wait as long as the other MPI functions that arrived earlier, but had to wait for the other ranks to finally arrive.

8.3 MPI Tracing Experiments (mpip)

8.3.1 MPI Tracing Experiments (mpip) performance data gathering

Much of this information is described in the main MPI Tracing Experiments section (above), but for completeness, here is the convenience script description for running the MPI-specific (mpi, mpit, mpip) tracing experiments.

```
> ossmpip “srun -N 4 -n 32 smg2000 -n 50 50 50” [default | <list MPI functions> | mpi_category]
```

If users give the mpi_category or a list of categories to the ossmpi command, then only those MPI functions corresponding to that category or categories will be traced. This table defines the MPI categories:

MPI Category	Argument
All MPI Functions	all
Collective Communicators	collective_com
Persistent Communicators	persistent_com
Synchronous Point to Point	synchronous_p2p
Asynchronous Non-Blocking	async_nonblocking
Asynchronous Point to Point	asynchronous_p2p
Process Topologies	process_topologies
Groups Contexts Communicators	graphs_contexts_comms
Environment	environment
Datatypes	datatypes
MPI File I/O	fileio

8.3.2 Viewing MPI Tracing Experiments (mpip) performance data via CLI

To launch the CLI on any experiment, use “openss -cli -f <database name>”.

This table describes the header and column data definitions for the default MPI experiment views.

Column Name	Column Definition
Exclusive MPI Call Time	Aggregated total exclusive time spent in the MPI function corresponding to this row of data.
% of MPI Time	Percentage of exclusive MPI time spent in the MPI function corresponding to this row of data relative to the total MPI time for all the MPI functions.
Number of Calls	Total number of calls to the MPI function corresponding to this row of data.
Min MPI Call Time Across Ranks	The minimum time that a rank or ranks, across all ranks, spent in the corresponding MPI function.
Rank of Min	The number of the rank that had the minimum time spent in the MPI function across all the ranks of the application.
Max MPI Call Time Across Ranks	The maximum time that a rank or ranks, across all ranks, spent in the corresponding MPI function.
Rank of Max	The number of the rank that had the maximum time spent in the MPI function across all the ranks of the application.
Average MPI Call Time Across Ranks	The average for the default view is the total amount of time for all the calls to a function divided by the total number of ranks. Thus, it is the average time that each rank spends in the function. As such, it is comparable to the Max and Min of a rank that is in the same report.

This is an example of the CLI default view for the MPI (mpip) experiments:

```

jsg@localhost:~/OpenSpeedShop_ROOT
File Edit View Search Terminal Help
openss>>expview
Exclusive MPI call % of Number Min Rank Max Rank Average Function (defining location)
times in Exclusive Total of Exclusive of Exclusive of Exclusive
seconds, CPU Time Calls Time Across Min Time Across Max Time Across
Ranks(s) Ranks(s) Ranks(s) Ranks(s) Ranks(s)
95034.514126 48.081511 27 3519.012910 23 3520.382349 6 3519.796819 PMPI_Init (libmonitor.so.0.0.0: pmpi.c,104)
48459.901000 24.517675 38907 380.920791 23 8518.191164 0 1794.811148 PMPI_Waitall (libmpich.so.1.0: waitall.c,57)
40505.333098 20.493161 12933 88.776030 16 3793.478559 0 1500.197522 MPI_Allreduce (libmpich.so.1.0: allreduce.c,59)
12065.686319 6.104483 279676 33.093728 26 1061.876716 24 446.877345 PMPI_Wait (libmpich.so.1.0: wait.c,51)
1408.907554 0.712819 279676 11.513939 0 147.764633 14 52.181761 PMPI_Isend (libmpich.so.1.0: isend.c,58)
178.581476 0.090351 279676 4.602243 26 10.555328 13 6.614129 PMPI_Irecv (libmpich.so.1.0: irecv.c,48)
openss>>

```

Here is an example of the CLI load balance view for the MPI (mpip) experiment. This view shows the minimum, maximum and average time per rank for each function and the rank that represents the maximum time and minimum time. Note that there may be more ranks that have the same maximum and minimum time per rank:

```

jsg@localhost:~/OpenSpeedShop_ROOT
File Edit View Search Terminal Help
openss>>expview -m loadbalance
Max Rank Min Rank Average Function (defining location)
Exclusive of Exclusive of Exclusive
Time Across Max Time Across Min Time Across
Ranks(s) Ranks(s) Ranks(s)
8518.191164 0 380.920791 23 1794.811148 PMPI_Waitall (libmpich.so.1.0: waitall.c,57)
3793.478559 0 88.776030 16 1500.197522 MPI_Allreduce (libmpich.so.1.0: allreduce.c,59)
3520.382349 6 3519.012910 23 3519.796819 PMPI_Init (libmonitor.so.0.0.0: pmpi.c,104)
1061.876716 24 33.093728 26 446.877345 PMPI_Wait (libmpich.so.1.0: wait.c,51)
147.764633 14 11.513939 0 52.181761 PMPI_Isend (libmpich.so.1.0: isend.c,58)
10.555328 13 4.602243 26 6.614129 PMPI_Irecv (libmpich.so.1.0: irecv.c,48)
openss>>

```

Here is an example of the ability to compare performance information at the rank level in the CLI. This example shows a comparison on the exclusive time metric for rank 0 and rank 23. These ranks were shown to be the ones with the maximum and minimum values for MPI_Waitall above. Users also could use the expview -r 0 and expview -r 23 to see times for just those ranks:

```

jsg@localhost:~/OpenSpeedShop_ROOT
File Edit View Search Terminal Help
openss>>expcompare -r0,23 -mtime
-r 0, -r 23, Function (defining location)
Exclusive MPI call Exclusive MPI call
times in times in
seconds, seconds,
8518.191164 380.920791 PMPI_Waitall (libmpich.so.1.0: waitall.c,57)
3793.478559 572.198026 MPI_Allreduce (libmpich.so.1.0: allreduce.c,59)
3519.065395 3519.012910 PMPI_Init (libmonitor.so.0.0.0: pmpi.c,104)
699.983590 363.937877 PMPI_Wait (libmpich.so.1.0: wait.c,51)
11.513939 28.868632 PMPI_Isend (libmpich.so.1.0: isend.c,58)
5.608623 5.892431 PMPI_Irecv (libmpich.so.1.0: irecv.c,48)
openss>>

```

This shows the top two call paths in the program that took the most time (with respect to MPI function calls):

```

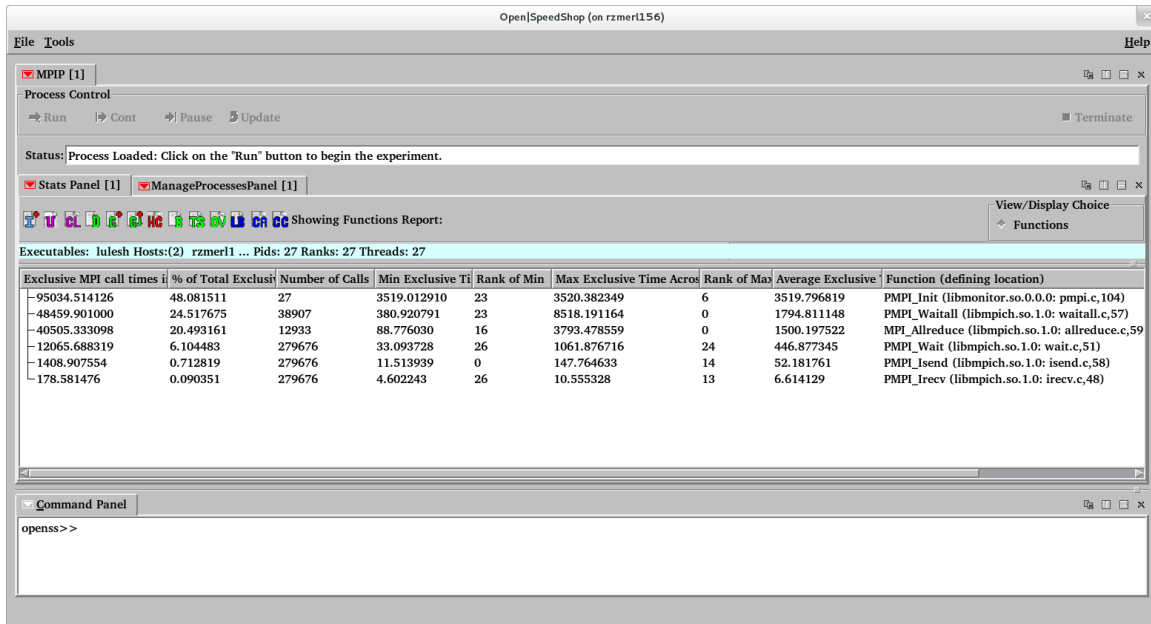
jsg@localhost:~/OpenSpeedShop_ROOT
File Edit View Search Terminal Help
openss>>expview -v calltrees,fullstack mpip2
Exclusive MPI call % of Number Min Rank Max Rank Average Call Stack Function (defining location)
times in Exclusive Total of Exclusive of Exclusive of Exclusive
seconds, CPU Time Calls Time Across Min Time Across Max Time Across
Ranks(s) Ranks(s) Ranks(s)
95034.514126 48.081511 27
> @ 562 in __libc_start_main (libmonitor.so.0.0.0: main.c,541)
>> @ 258 in __libc_start_main (libc-2.12.so: libc-start.c,96)
>>> @ 517 in monitor_main (libmonitor.so.0.0.0: main.c,492)
>>>> @ 5340 in main (lulesh: lulesh.cc,5333)
>>>> @ 104 in PMPI_Init (libmonitor.so.0.0.0: pmpi.c,104)
40505.333098 20.493161 12933
> @ 562 in __libc_start_main (libmonitor.so.0.0.0: main.c,541)
>> @ 258 in __libc_start_main (libc-2.12.so: libc-start.c,96)
>>> @ 517 in monitor_main (libmonitor.so.0.0.0: main.c,492)
>>>> @ 2381 in main (lulesh: lulesh.cc,5333)
>>>> @ 59 in MPI_Allreduce (libmpich.so.1.0: allreduce.c,59)
openss>>

```

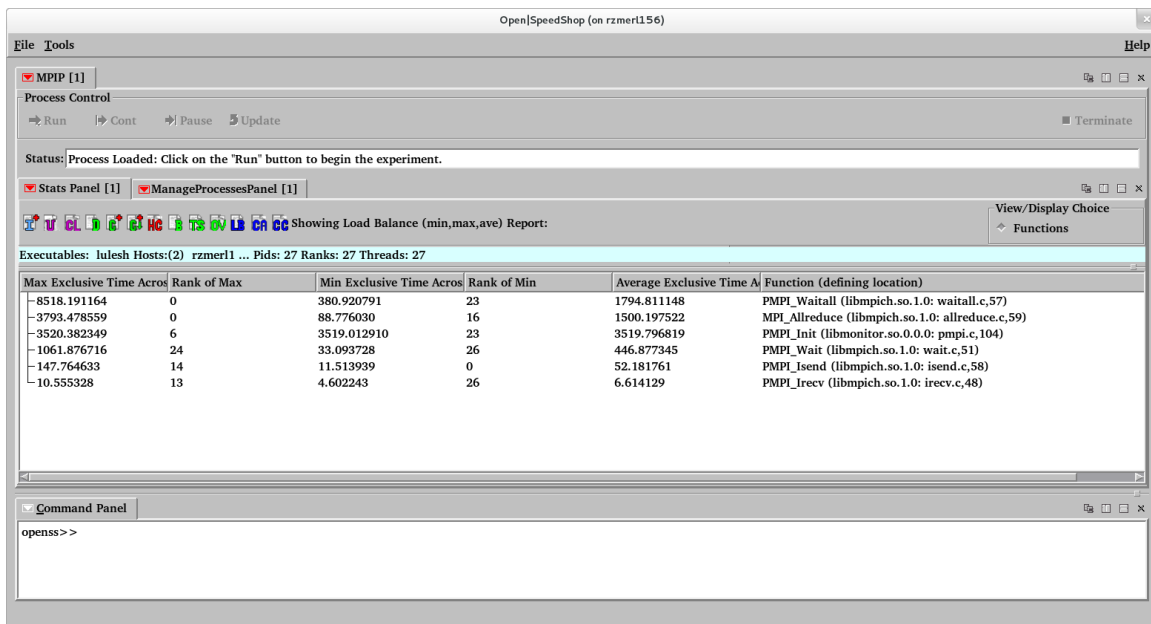

8.3.3 MPI Viewing Tracing Experiments (mpip) performance data via GUI

To launch the GUI on any experiment, use “openss -f <database name>”.

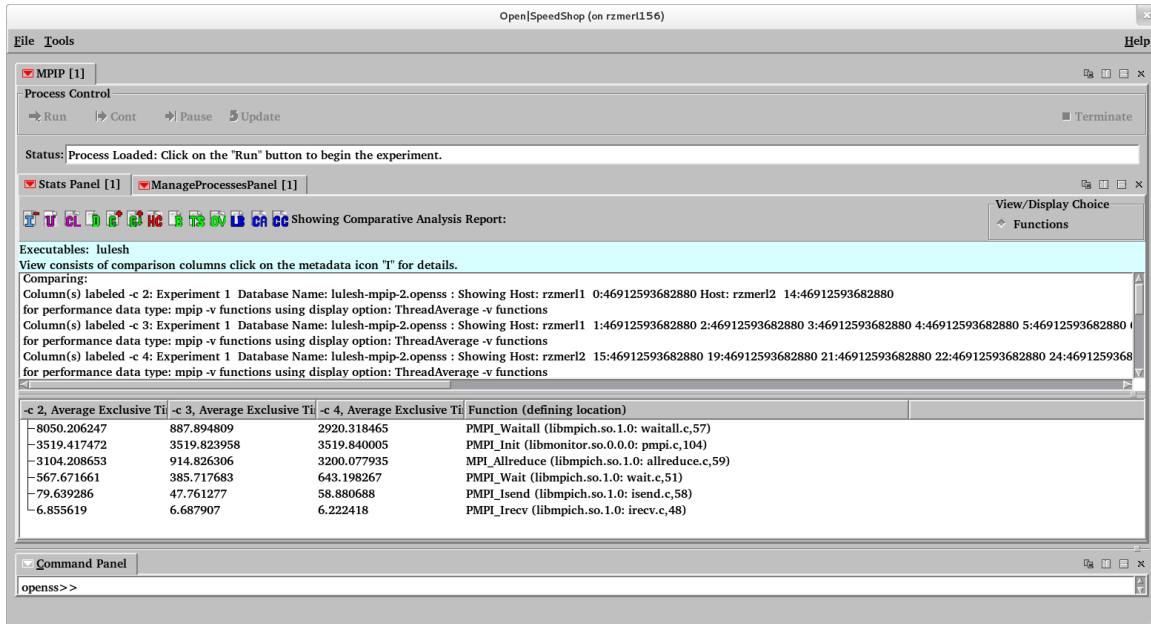
This is an example of the GUI default view for the MPI (mpip) experiment:



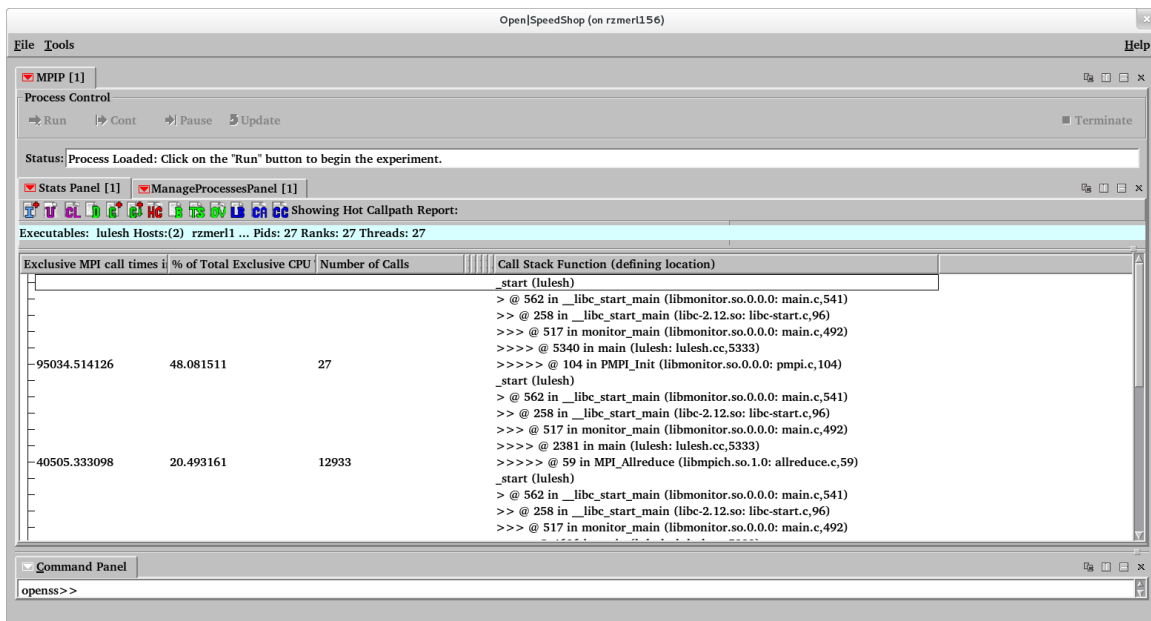
This shows the load balance for this execution of lulesh on 27 ranks:



This shows the cluster analysis view for this run of lulesh on 27 ranks. This view groups similarly performing ranks to help users locate groups of ranks that are outliers with respect to the other ranks.



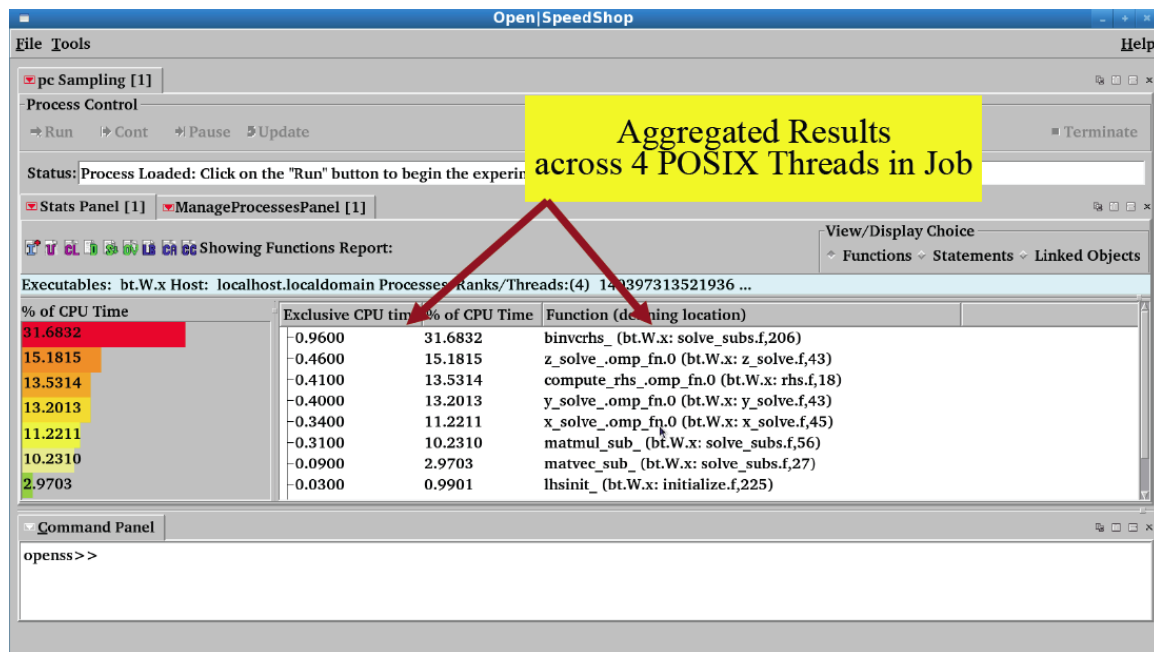
This shows the hot call paths in the application:



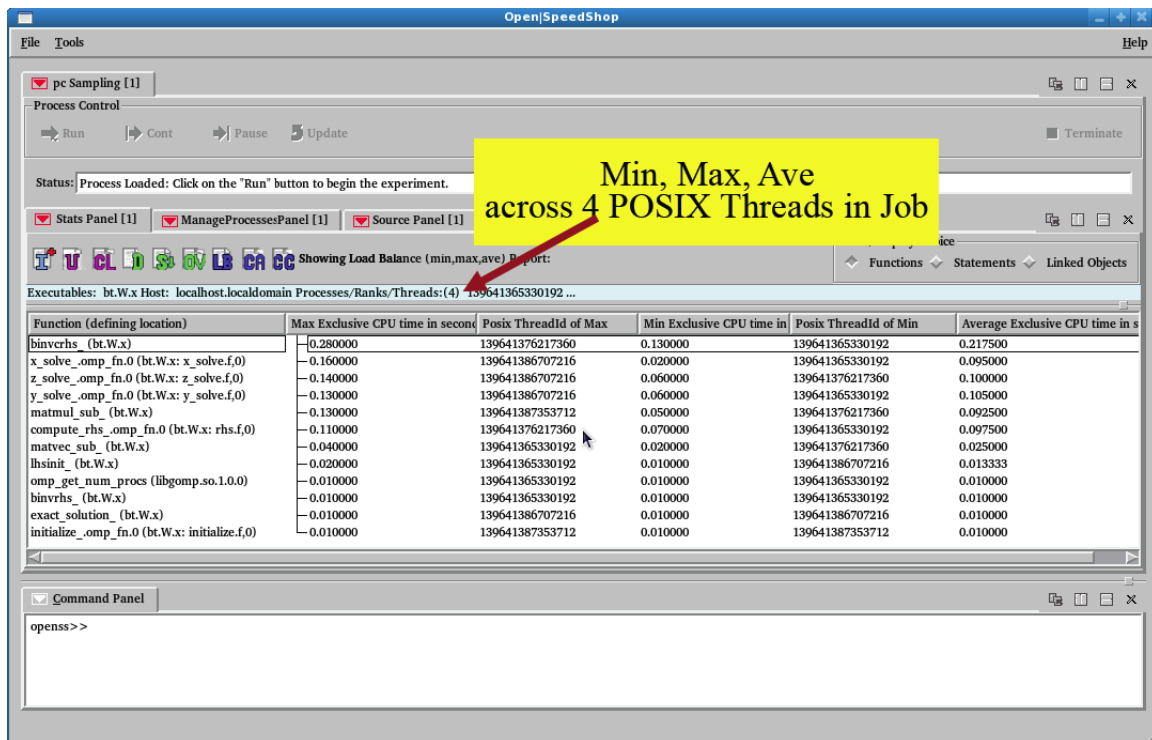
9 Threading Analysis Section

In the previous sections we described experiments that use MPI, but users can do a similar analysis on applications that use threads. To analyze a threaded application, users can first run the pcsamp experiment to get an overview, then check the load balance view to detect any widely varying values, and finally do cluster analysis to find outliers.

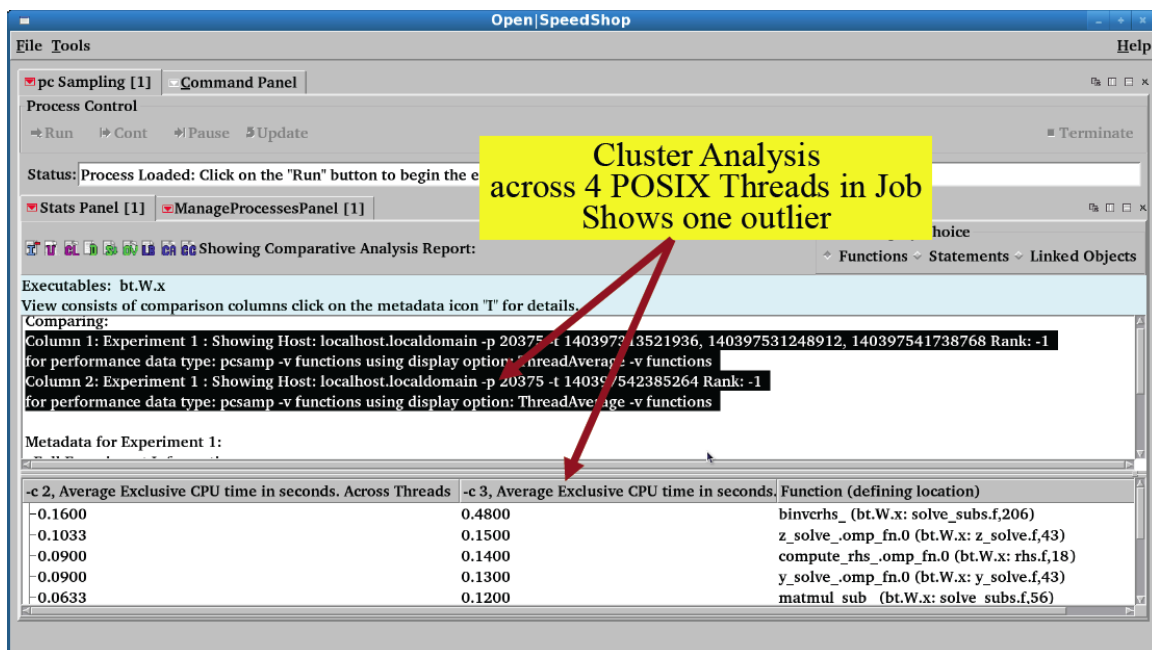
This shows the default view for an application with four threads. The information displayed is the aggregated total from all threads:



This is the load balance view based on functions:



Next is a cluster analysis view based on functions:



9.1 Threading Specific Experiment (pthreads)

O|SS also has available an experiment specific to tracking and analyzing POSIX thread function calls. The experiment, pthreads, traces several POSIX thread-related

functions. Like all the other tracing experiments, it provides the number of calls, time spent in each function, call paths to each POSIX thread function and an event-by-event trace. Load balance and cluster analysis features also are available.

9.1.1 Threading Specific (pthreads) experiment performance data gathering (oss pthreads)

To run the pthreads experiment, use the oss pthreads convenience script while placing how the application would normally run in quotes, as shown here:

```
oss pthreads "mpirun -np 4 ./smg2000 -n 15 15 15"
[openss]: pthreads using default experiment trace function list.
Creating topology file for frontend host localhost
Generated topology file: ./cbtfAutoTopology
Running pthreads collector.
Program: mpirun -np 4 ./smg2000 -n 15 15 15
Number of mrnet backends: 4
Topology file used: ./cbtfAutoTopology
executing mpi program: mpirun -np 4 cbtfrun --mpi --mrnet -c pthreads ./smg2000 -n 15 15 15
Running with these driver parameters:
  (nx, ny, nz) = (15, 15, 15)
  (Px, Py, Pz) = (4, 1, 1)
  (bx, by, bz) = (1, 1, 1)
  (cx, cy, cz) = (1.000000, 1.000000, 1.000000)
  (n_pre, n_post) = (1, 1)
  dim = 3
  solver ID = 0
=====
Struct Interface:
=====
Struct Interface:
  wall clock time = 0.000475 seconds
  cpu clock time = 0.000000 seconds
=====
Setup phase times:
=====
SMG Setup:
  wall clock time = 0.047075 seconds
  cpu clock time = 0.050000 seconds
=====
Solve phase times:
=====
SMG Solve:
  wall clock time = 0.092030 seconds
  cpu clock time = 0.100000 seconds

Iterations = 5
Final Relative Residual Norm = 6.027844e-07

All Threads are finished.
default view for /home/fred/DEMOS/demos/mpi/openmpi-1.8.2/smg2000/test/smg2000-
pthreads-1.openss
[openss]: The restored experiment identifier is: -x 1
```

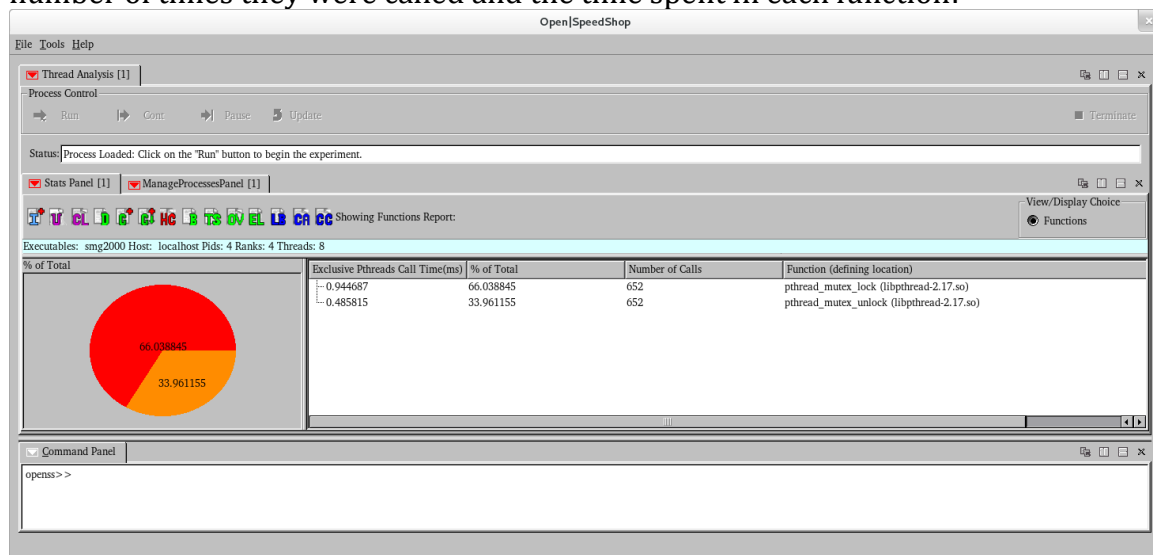
Performance data spans 0.881730 ms from 2017/01/04 19:14:25 to 2017/01/04 19:14:26

Exclusive Pthreads Call Time(ms)	% of Total Calls	Number of	Function (defining location)
0.944687	66.038845	652	pthread_mutex_lock (libpthread-2.17.so)
0.485815	33.961155	652	pthread_mutex_unlock (libpthread-2.17.so)

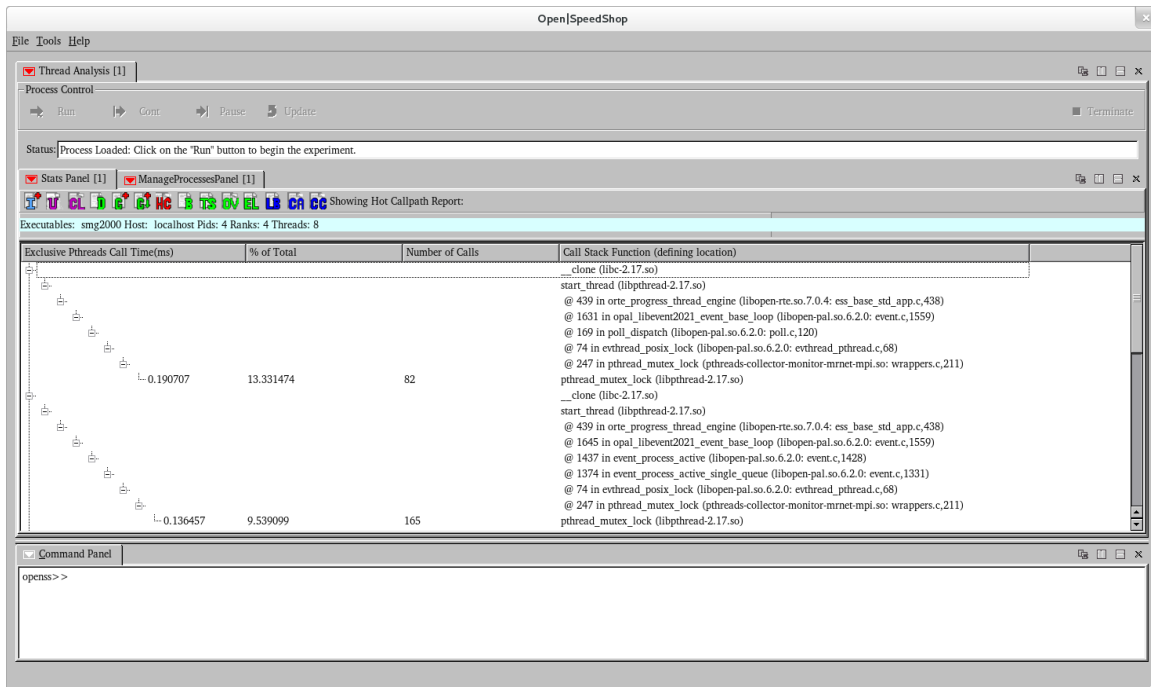
9.1.2 Viewing Threading Specific (pthreads) experiment performance data via GUI

To launch the GUI on any experiment, use “openss -f <database name>”.

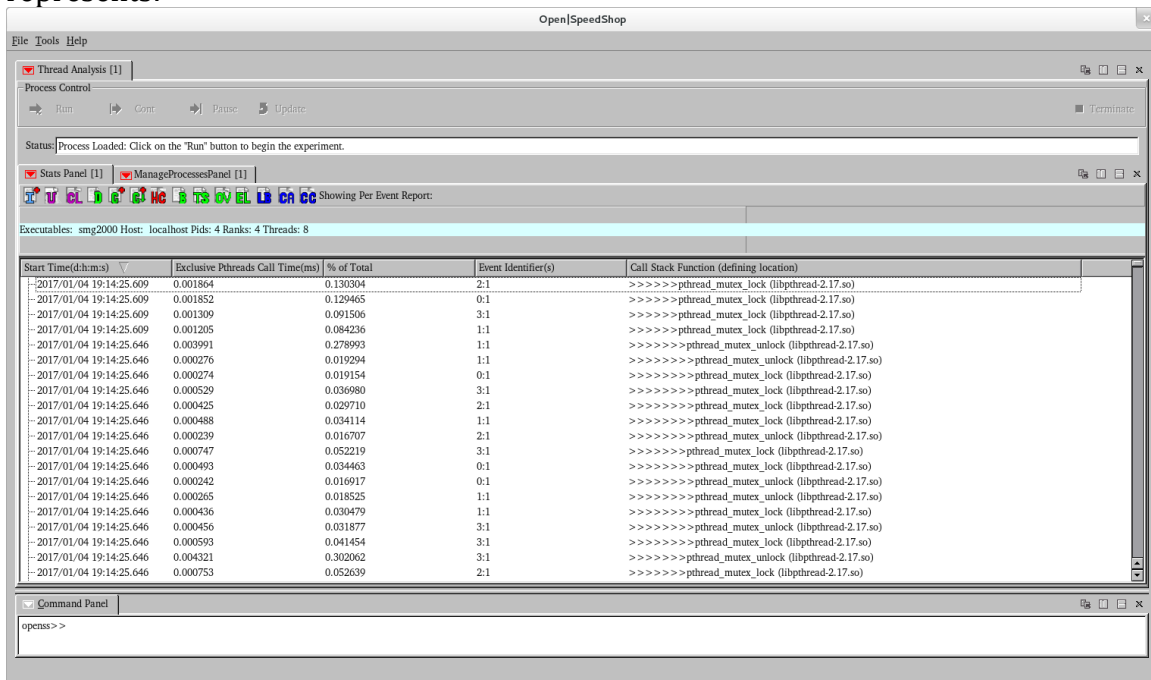
Here are three pthreads experiment views. First is the default, listing the POSIX thread function routines that were called in the application being monitored, the number of times they were called and the time spent in each function:



This shows the top five time-consuming POSIX thread function call paths through the application:



Last is an event list view, showing POSIX thread function calls in the order they occurred with the rank and thread the call originated from, the time spent in the POSIX thread function call event and the percentage of the total time that represents:



9.1.3 Viewing Threading Specific (pthreads) experiment performance data via CLI

To launch the CLI on any experiment, use “`openss -cli -f <database name>`”.

```
openss>>[openss]: The restored experiment identifier is: -x 1
openss>>expview
```

Exclusive Pthreads Call Time(ms)	% of Total Calls	Number of	Function (defining location)
0.944687	66.038845	652	pthread_mutex_lock (libpthread-2.17.so)
0.485815	33.961155	652	pthread_mutex_unlock (libpthread-2.17.so)

```
openss>>expview -m loadbalance
```

Max Rank Exclusive Pthreads call time in seconds. Across Ranks(ms)	Min Rank of Exclusive Pthreads call time in seconds. Across Ranks(ms)	Average of Exclusive Min Pthreads call time in seconds. Across Ranks(ms)	Function (defining location)
0.165071	2	0.065597	1 0.118086 pthread_mutex_lock (libpthread-2.17.so)
0.125825	1	0.007971	2 0.060727 pthread_mutex_unlock (libpthread-2.17.so)

```
openss>>expview -m loadbalance -r 0
```

Max ThreadId Exclusive Pthreads call time in seconds. Across ThreadIds(ms)	Min ThreadId Exclusive Pthreads call time in seconds. Across ThreadIds(ms)	Average Function (defining location)
0.162899	1	0.071457 0 0.117178 pthread_mutex_lock (libpthread-2.17.so)
0.120930	1	0.009044 0 0.064987 pthread_mutex_unlock (libpthread-2.17.so)

```
openss>>expcompare -r1 -t0:1
```

-t 0 -r 1, 1, % of Exclusive Pthreads Call Time(ms)	-t 0 -r 1, 1, % of Total Number of Pthreads Calls Time(ms)	-t 1 -r 1, 1, % of Exclusive Pthreads Call Time(ms)	-t 1 -r 1, 1, % of Total Number of Pthreads Calls Time(ms)	Function (defining location)
0.065597	87.269510	24	0.161611	56.225038 137 pthread_mutex_lock (libpthread-2.17.so)
0.009569	12.730490	24	0.125825	43.774962 137 pthread_mutex_unlock (libpthread-2.17.so)

```
openss>>
```

```
openss>>expview -vfullstack pthreads4
```

Exclusive Pthreads Call Time(ms)	% of Total Calls	Number of	Call Stack Function (defining location)
			__clone (libc-2.17.so)
			>start_thread (libpthread-2.17.so)
			>> @ 439 in orte_progress_thread_engine (libopen-rte.so.7.0.4: ess_base_std_app.c,438)
			>>> @ 1631 in opal_libevent2021_event_base_loop (libopen-pal.so.6.2.0: event.c,1559)
			>>>> @ 169 in poll_dispatch (libopen-pal.so.6.2.0: poll.c,120)
			>>>>> @ 74 in evthread_posix_lock (libopen-pal.so.6.2.0: evthread_pthread.c,68)
			>>>>>> @ 247 in pthread_mutex_lock (pthreads-collector-monitor-mrnet-mpi.so:


```

wrappers.c,211)
0.190707 13.331474 82 >>>>>>>pthread_mutex_lock (libpthread-2.17.so)
    _clone (libc-2.17.so)
    >start_thread (libpthread-2.17.so)
    >> @ 439 in orte_progress_thread_engine (libopen-rte.so.7.0.4: ess_base_std_app.c,438)
    >>> @ 1645 in opal_libevent2021_event_base_loop (libopen-pal.so.6.2.0: event.c,1559)
    >>>> @ 1437 in event_process_active (libopen-pal.so.6.2.0: event.c,1428)
    >>>>> @ 1374 in event_process_active_single_queue (libopen-pal.so.6.2.0: event.c,1331)
    >>>>>> @ 74 in evthread_posix_lock (libopen-pal.so.6.2.0: evthread_pthread.c,68)
    >>>>>>> @ 247 in pthread_mutex_lock (pthreads-collector-monitor-mrnet-mpi.so:
wrappers.c,211)
0.136457 9.539099 165 >>>>>>>pthread_mutex_lock (libpthread-2.17.so)
    _clone (libc-2.17.so)
    >start_thread (libpthread-2.17.so)
    >> @ 439 in orte_progress_thread_engine (libopen-rte.so.7.0.4: ess_base_std_app.c,438)
    >>> @ 1631 in opal_libevent2021_event_base_loop (libopen-pal.so.6.2.0: event.c,1559)
    >>>> @ 165 in poll_dispatch (libopen-pal.so.6.2.0: poll.c,120)
    >>>>> @ 81 in evthread_posix_unlock (libopen-pal.so.6.2.0: evthread_pthread.c,78)
    >>>>>> @ 298 in pthread_mutex_unlock (pthreads-collector-monitor-mrnet-mpi.so:
wrappers.c,262)
0.085655 5.987758 82 >>>>>>>pthread_mutex_unlock (libpthread-2.17.so)
    _clone (libc-2.17.so)
    >start_thread (libpthread-2.17.so)
    >> @ 439 in orte_progress_thread_engine (libopen-rte.so.7.0.4: ess_base_std_app.c,438)
    >>> @ 1645 in opal_libevent2021_event_base_loop (libopen-pal.so.6.2.0: event.c,1559)
    >>>> @ 1437 in event_process_active (libopen-pal.so.6.2.0: event.c,1428)
    >>>>> @ 1367 in event_process_active_single_queue (libopen-pal.so.6.2.0: event.c,1331)
    >>>>>> @ 81 in evthread_posix_unlock (libopen-pal.so.6.2.0: evthread_pthread.c,78)
    >>>>>>> @ 298 in pthread_mutex_unlock (pthreads-collector-monitor-mrnet-mpi.so:
wrappers.c,262)
0.085246 5.959167 105 >>>>>>>pthread_mutex_unlock (libpthread-2.17.so)
openss>>

```

9.2 OpenMP Related Performance Analysis

9.2.1 OpenMP Thread Wait Detection using OMPT interface

If built with the OMPT enhanced OpenMP runtime library, O|SS will detect OpenMP thread wait time. In general, OpenMP support in O|SS is available in two forms: augmenting the sampling experiments and providing an OpenMP specific experiment.

9.2.1.1 Augmentation of O|SS sampling experiments

The first form integrates information gathered from the OpenMP runtime through the new OMPT tools interface into existing displays and experiments. This is done by aggregating the information from the runtime into “pseudo functions” and listing them as part of the standard profile (without any details of what is actually executed in the runtime). Here’s an example showing thread idle time (as part of the pseudo function **IDLE**) and barrier time (as part of **WAIT_BARRIER**). Other states in the runtime would be shown similarly:

```
openss>>expviewr11SEP
```

Exclusive % of Function (defining location)	CPU time	CPU in Time	seconds
453.0900	14.4423	CalcFBHourglassForceForElems()	(lulesh2.0: lulesh.cc,745)
325.5600	10.3773	IntegrateStressForElems()	(lulesh2.0: lulesh.cc,526)
312.5800	9.9635	EvalEOSForElems(Domain&, double*, int, int*, int)	(lulesh2.0: lulesh.cc,2236)
306.6100	9.7732	LagrangeNodal(Domain&)	(lulesh2.0: lulesh.cc,1253)
230.6600	7.3523	CalcKinematicsForElems(Domain&, double*, double, int)	(lulesh2.0: lulesh.cc,1535)
160.3400	5.1109	IDLE	(pcsamp-collector-monitor-mrnet-mpi.so: collector.c,477)
156.6800	4.9942	psm_mq_peek	(libpsm_infinipath.so.1.14)
150.4100	4.7943	ips_ptl_poll	(libpsm_infinipath.so.1.14)
132.9100	4.2365	CalcElemVolumeDerivative(double*, double*, double*, double const*, double const*, double const*)	(lulesh2.0: lulesh.cc,658)
105.9600	3.3775	CalcMonotonicQGradientsForElems(Domain&, double*)	(lulesh2.0: lulesh.cc,1643)
99.8600	3.1831	__pthread_cond_signal	(libpthread-2.12.so: pthread_cond_signal.S,38)
77.6300	2.4745	__GI_vfprintf	(libc-2.12.so: vfprintf.c,201)
77.2600	2.4627	sbrk	(libc-2.12.so: sbrk.c,35)
60.2000	1.9189	CalcMonotonicQRegionForElems(Domain&, int, double*, double)	(lulesh2.0: lulesh.cc,1792)
41.7800	1.3317	main	(lulesh2.0: lulesh.cc,2690)
34.1000	1.0869	WAIT_BARRIER	(pcsamp-collector-monitor-mrnet-mpi.so: collector.c,501)
30.6000	0.9754	__psmi_poll_internal	(libpsm_infinipath.so.1.14)
25.2300	0.8042	_IO_default_xsputn_internal	(libc-2.12.so: genops.c,452)

What does using the OMPT interface in O|SS let users do?

O|SS applies the OMPT API blame callbacks for `ompt_event_thread_idle`, `ompt_event_thread_barrier` and `ompt_event_thread_wait_barrier` to samples taken in the OpenMP library that otherwise would be shown as `__kmp_barrier`, `__kmp_wait_sleep`, etc. in the Intel libiomp5 library. O|SS uses the libiomp5 library with the OMPT API enabled at runtime to do this for all OpenMP codes run with the pcsamp, usertime and hardware counter-based experiments. The user can then see the sample time per thread for idle, barrier and wait_barrier. The user also can use the loadbalance metric to see the min, max and average of these blame events or use the expcompare across all threads to compare individual metrics.

For barrier symbols, samples taken when a thread is waiting at a barrier are inclusive to total barrier time; that is, adding barrier and wait_barrier metrics gives total barrier time.

Essentially these blame metrics, as used in the O|SS sampling experiments, provide the time a thread is idle and the time spent at a barrier (including waiting at a barrier).

Adding barrier and wait_barrier gives the total samples taken at a barrier; wait_barrier is just the samples within that barrier when there is a thread_barrier_wait condition.

Here is an example from the lulesh sequential OpenMP case:

```

openss>>expview -f OMPT* -m time
Exclusive Function (defining location)
CPU time
  in
seconds.
```

```

1.460000 IDLE (pcsamp-collector-monitor-mrnet.so: collector.c,99)
0.470000 WAIT_BARRIER (pcsamp-collector-monitor-mrnet.so: collector.c,129)
0.020000 BARRIER (pcsamp-collector-monitor-mrnet.so: collector.c,113)

openss>>expcompare -f OMPT* -m time -t0:4

-t 0, -t 2, -t 3, -t 4, Function (defining location)
Exclusive Exclusive Exclusive Exclusive
CPU time CPU time CPU time CPU time
in in in in
seconds. seconds. seconds. seconds.
0.360000 0.030000 0.070000 0.010000 WAIT_BARRIER (pcsamp-collector-monitor-mrnet.so:
collector.c,129)
0.000000 0.500000 0.400000 0.560000 IDLE (pcsamp-collector-monitor-mrnet.so:
collector.c,99)
0.000000 0.000000 0.010000 0.010000 BARRIER (pcsamp-collector-monitor-mrnet.so:
collector.c,113)
openss>>expview -f OMPT* -m time -t3

Exclusive Function (defining location)
CPU time
in
seconds.
0.400000 IDLE (pcsamp-collector-monitor-mrnet.so: collector.c,99)
0.070000 WAIT_BARRIER (pcsamp-collector-monitor-mrnet.so: collector.c,129)
0.010000 BARRIER (pcsamp-collector-monitor-mrnet.so: collector.c,113)

```

This shows that most of the barrier samples were taken when the thread was waiting at the barrier. The BARRIER and WAIT_BARRIER symbols replace samples that would have appeared as `__kmp_barrier` (or possibly `__kmp_join_barrier`) in the latest libiomp5. The IDLE samples replace `__kmp_wait_sleep` in the real libiomp5.

The above applies to pcsamp, usertime and the three hardware counter collectors (hwc, hwctime, and hwcsamp), is essentially telling the user the time a thread is idle and the time spent at a barrier (including waiting at a barrier).

From the example above, with the addition of BARRIER, it can be inferred that most of the barrier time for thread 2 was spent waiting at the barrier.

The usertime experiment on an OpenMP application can help pinpoint the origin of wait barrier time in the source. For example:

```

openss>>expview
Exclusive Inclusive % of Function (defining location)
CPU time CPU time Total
in in Exclusive
seconds. seconds. CPU Time
23.200000 23.200000 38.648263 IDLE (usertime-collector-monitor-mrnet.so: collector.c,122)
13.142857 13.142857 21.894336 MAIN_.omp_fn.2 (stress_omp: stress_omp.f,179)
12.885714 12.885714 21.465969 MAIN_.omp_fn.5 (stress_omp: stress_omp.f,227)
4.742857 4.742857 7.901000 WAIT_BARRIER (usertime-collector-monitor-mrnet.so: collector.c,150)
2.000000 11.771428 3.331747 MAIN_ (stress_omp: stress_omp.f,1)
1.257143 1.257143 2.094241 __kernel_cosf (libm-2.12.so: k_cosf.c,45)

```

This shows the call path that points to the source lines leading to the thread waiting in the barrier:

```

openss>>expview -vfullstack -f WAIT_BARRIER usertime1

Exclusive Inclusive   % of Call Stack Function (defining location)
CPU time  CPU time    Total
in        in        Exclusive
seconds.  seconds.  CPU Time
      _start (stress_omp)
      > @ 556 in __libc_start_main (libmonitor.so.0.0.0: main.c,541)
      >> __libc_start_main (libc-2.12.so)
      >>> @ 517 in monitor_main (libmonitor.so.0.0.0: main.c,492)
      >>>> main (stress_omp)
      >>>>> @ 227 in MAIN_ (stress_omp: stress_omp.f,1)
      >>>>>> @ 557 in __kmp_api_GOMP_parallel_end_10_alias (libiomp5.so: kmp_gsupport.c,490)
      >>>>>>> @ 2395 in __kmp_join_call (libiomp5.so: kmp_runtime.c,2325)
      >>>>>>>> @ 7114 in __kmp_internal_join (libiomp5.so: kmp_runtime.c,7093)
      >>>>>>>>> @ 1458 in __kmp_join_barrier(int) (libiomp5.so: kmp_barrier.cpp,1371)
1.742857 1.742857 2.903379 >>>>>>>>>> @ 150 in WAIT_BARRIER (usertime-collector-monitor-mrnet.so:
collector.c,150)

```

These changes mean O|SS now has support in:

- Added idle, barrier, and wait_barrier blame support to all sampling collectors.
- Improved naming (IDLE, BARRIER, WAIT_BARRIER) for sample call site.
- Support for building from LLVM OpenMP.
- Updated runtime codes to allow running OMPT with gnu/gomp binaries (which means libiomp5 replaces libgomp since gomp does not support OMPT directly).

Also, note that the OMPT aspect of OSS works with gcc or g++ generated OpenMP code (and likely clang). No matter which compiler was used to generate the OpenMP code, the Intel libiomp5.so runtime is used for its OMPT API.

With the current OMPT support, no distinctions are made specific to the idle time or wait_barrier time subcategorization.

Looking at the code in question as found in kmp_runtime.c, __kmp_launch_thread has a while loop, that sets the ompt state to ompt_state_idle, calls __kmp_fork_barrier, and then sets the ompt state to a default value of ompt_state_overhead. If __kmp_wait_sleep is called while ompt state is ompt_state_idle, then the OMPT API considers the thread "idle". In that case, would this "OMP_thread_idle" be considered "OMP_thread_fork_wait"?

The other case where __kmp_wait_sleep is entered, with an OMPT state of ompt_state_wait_barrier, is when that OMPT state is managed by the __kmp_barrier and code __kmp_join_barrier in kmp_runtime.c. The wait and join barrier are both using ompt_state_wait_barrier as coded by the OMPT interface. That means those

are combined in what O|SS reports for OMP_thread_wait_barrier in the pcsamp example, actually WAIT_BARRIER.

9.2.2 O|SS OpenMP specific profiling experiment (omptp)

The second form is a separate OpenMP specific profiling experiment (omptp).

9.2.2.1 OpenMP Specific (omptp) experiment performance data gathering (ossomptp)

To run the OpenMP specific experiment, use the ossomptp convenience script, placing how the application would normally be run in quotes:

```
export OMP_NUM_THREADS=4
ossomptp "mpirun -np 4 ./smg2000 -n 15 15 15"
```

9.2.2.2 Viewing OpenMP Specific (omptp) experiment performance data via GUI

TBD. The GUI does not currently support the omptp experiment outputs.

9.2.2.3 Viewing OpenMP Specific (omptp) experiment performance data via CLI

These three CLI examples show the most important ways to view OMPTP experiment data. The default view shows the timing of the parallel regions, idle, barrier, and wait barrier as an aggregate across all threads:

```
openss -cli -f ./matmult-omptp-0.openss
openss>>expview
```

Exclusive times in seconds.	Inclusive times in seconds.	% of Total Exclusive CPU Time	Function (defining location)
44.638794	45.255843	93.499987	compute_omp_fn.1 (matmult: matmult.c,68)
1.744841	1.775104	3.654726	compute_interchange_omp_fn.3 (matmult: matmult.c,118)
0.701720	0.701726	1.469817	compute_triangular_omp_fn.2 (matmult: matmult.c,95)
0.652438	0.652438	1.366591	IDLE (omptp-collector-monitor-mrnet.so: collector.c,573)
0.004206	0.009359	0.008810	initialize_omp_fn.0 (matmult: matmult.c,32)
0.000032	0.000032	0.000068	BARRIER (omptp-collector-monitor-mrnet.so: collector.c,587)
0.000000	0.000000	0.000001	WAIT_BARRIER (omptp-collector-monitor-mrnet.so: collector.c,602)

This example shows the comparison of exclusive time across all threads for the parallel regions, idle, barrier, and wait barrier:

```
openss>>expcompare -mtime -t0:4
```

-t 0,	-t 2,	-t 3,	-t 4, Function (defining location)
Exclusive times in	Exclusive times in	Exclusive times in	Exclusive times in

```
seconds. seconds. seconds. seconds.
11.313892 11.081346 11.313889 10.929668 compute_omp_fn.1 (matmult: matmult.c,68)
0.443713 0.430553 0.429635 0.440940 compute_interchange_omp_fn.3 (matmult: matmult.c,118)
0.253632 0.213238 0.164875 0.069975 compute_triangular_omp_fn.2 (matmult: matmult.c,95)
0.001047 0.001100 0.001095 0.000964 initialize_omp_fn.0 (matmult: matmult.c,32)
0.000008 0.000008 0.000006 0.000010 BARRIER (omptp-collector-monitor-mrnet.so:
collector.c,587)
0.000000 0.000000 0.000000 0.000000 WAIT_BARRIER (omptp-collector-monitor-mrnet.so:
collector.c,602)
0.000000 0.247592 0.015956 0.388890 IDLE (omptp-collector-monitor-mrnet.so: collector.c,573)
```

This example shows the load balance of time across all threads for the parallel regions, idle, barrier, and wait barrier:

```
openss>>expview -mloadbalance
```

Max Exclusive Time Across OpenMp ThreadIds(s)	OpenMp ThreadId	Min Exclusive Time Across OpenMp ThreadIds(s)	OpenMp ThreadId	Average Exclusive Time Across ThreadIds(s)	Function (defining location)
11.313892	0	10.929668	4	11.159699	compute_omp_fn.1 (matmult: matmult.c,68)
0.443713	0	0.429635	3	0.436210	compute_interchange_omp_fn.3 (matmult: matmult.c,118)
0.388890	4	0.015956	3	0.217479	IDLE (omptp-collector-monitor-mrnet.so: collector.c,573)
0.253632	0	0.069975	4	0.175430	compute_triangular_omp_fn.2 (matmult: matmult.c,95)
0.001100	2	0.000964	4	0.001052	initialize_omp_fn.0 (matmult: matmult.c,32)
0.000010	4	0.000006	3	0.000008	BARRIER (omptp-collector-monitor-mrnet.so: collector.c,587)
0.000000	0	0.000000	0	0.000000	WAIT_BARRIER (omptp-collector-monitor-mrnet.so: collector.c,602)

9.3 Hybrid (OpenMP and MPI) Performance Analysis

For this tutorial example, we ran O|SS convenience script on the AMG2013 hybrid program and created a database file that has eight ranks, each with four underlying OpenMP threads.

This example is designed to show that users can first examine hybrid performance at the MPI level, then go under the MPI rank to see how the threads are performing. At the MPI level, users can see load balance and outliers, then focus on a rank and look at load balance and outliers for the underlying threads. Within a terminal window, enter:

```
openss -f amg2013-pcsamp-2.openss
```

to bring up the O|SS GUI.

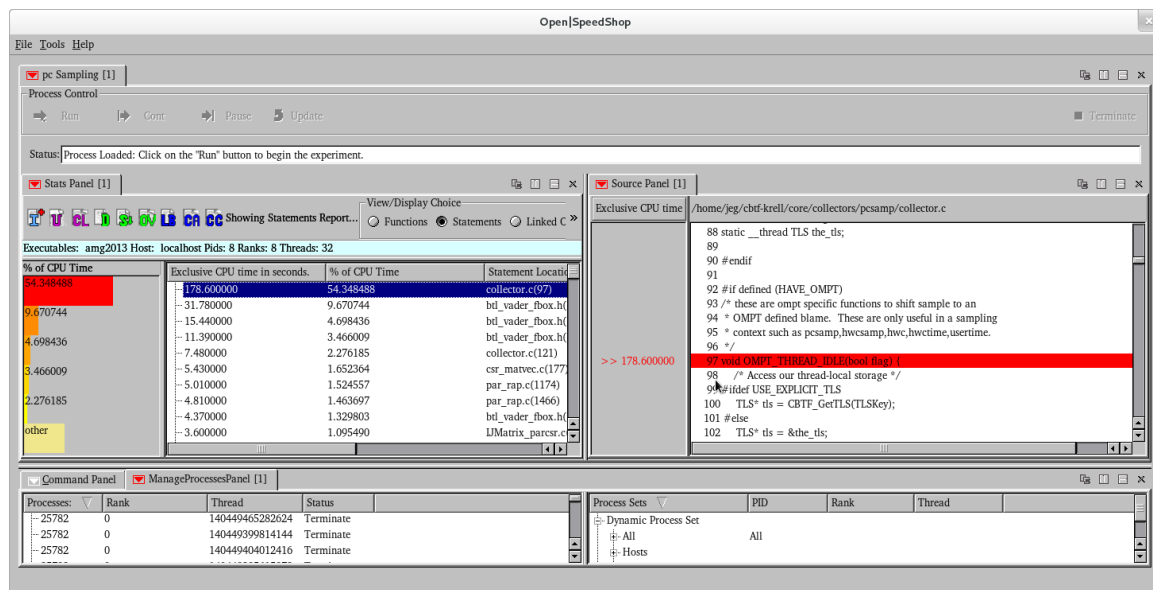
The GUI view below displays the aggregated results for the application at statement-level granularity. When the default view first comes up, it's at function-level

granularity. To switch to the statement level, select the Statements button in the View/Display Choice section on the right-hand side of the Stats Panel display and then click the “D” icon for default view. This will switch the Stats Panel view to statement-level granularity.

The Stats Panel now displays statements that took the most time in the application run. For this execution of AMG2013, the statement at line 97 of the OpenMP thread idle wrapper took the most time. This means that the most time spent in this run was a thread idle routine. This is expected because the ranks and number of threads in this run were both oversubscribed. Double clicking on the statement focuses OJSS on the source for that line of the application source and highlights that line.

In the view below, the ManageProcess panel tab is moved to the lower panel and the upper panel is split using the vertical splitter icon on the far-right side of the original upper panel.

Note: Left mouse down and hold on the panel tab then slide the panel to be moved to another location on the OJSS GUI or off onto other parts of the display.

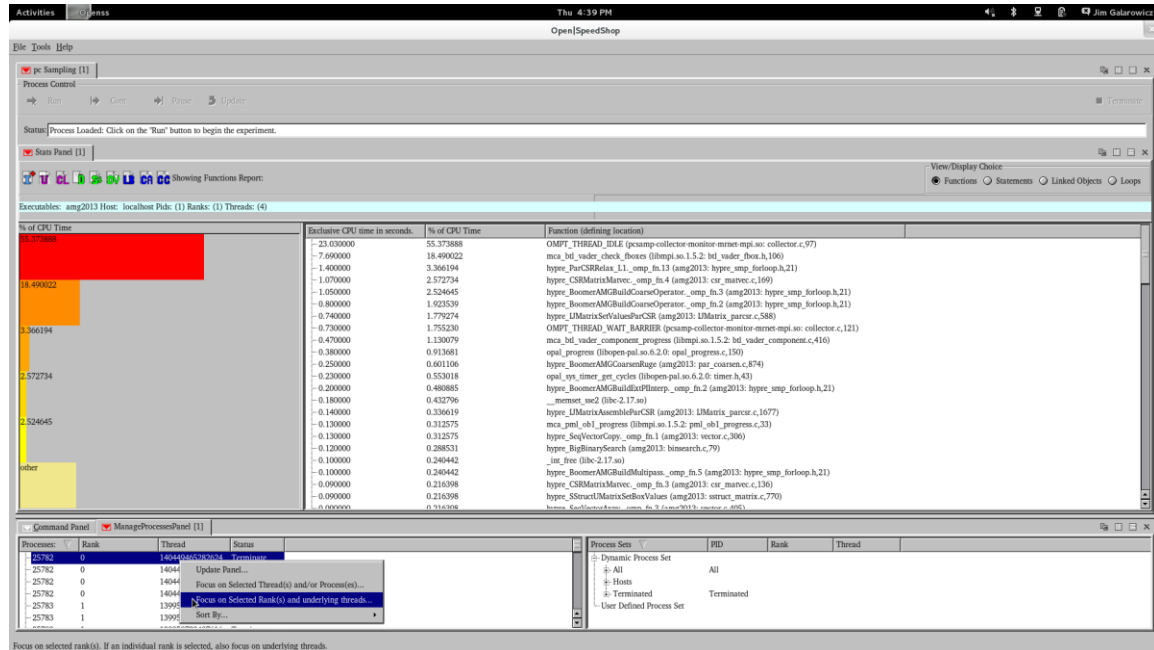


9.3.1 Focus on individual Rank to get Load Balance for Underlying Threads

The next view uses the ManageProcess panel to highlight one rank and an individual thread within the rank to show only that thread's performance data in the Stats Panel view. The existing view is the default function view with data aggregated across all ranks and threads for this run of the AMG2013 application.

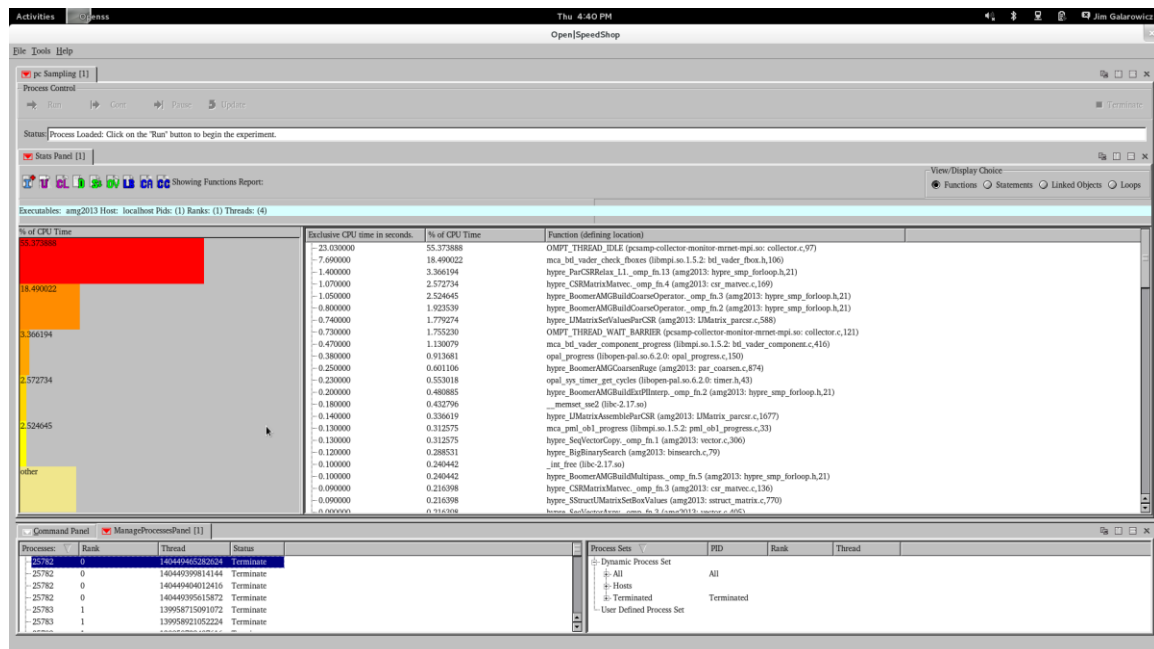
Note: Use the “focus on threads and processes” Manage Process panel option to focus Panel on individual threads within a rank. Right mouse button down on the Manage

Process panel tab to see the options.



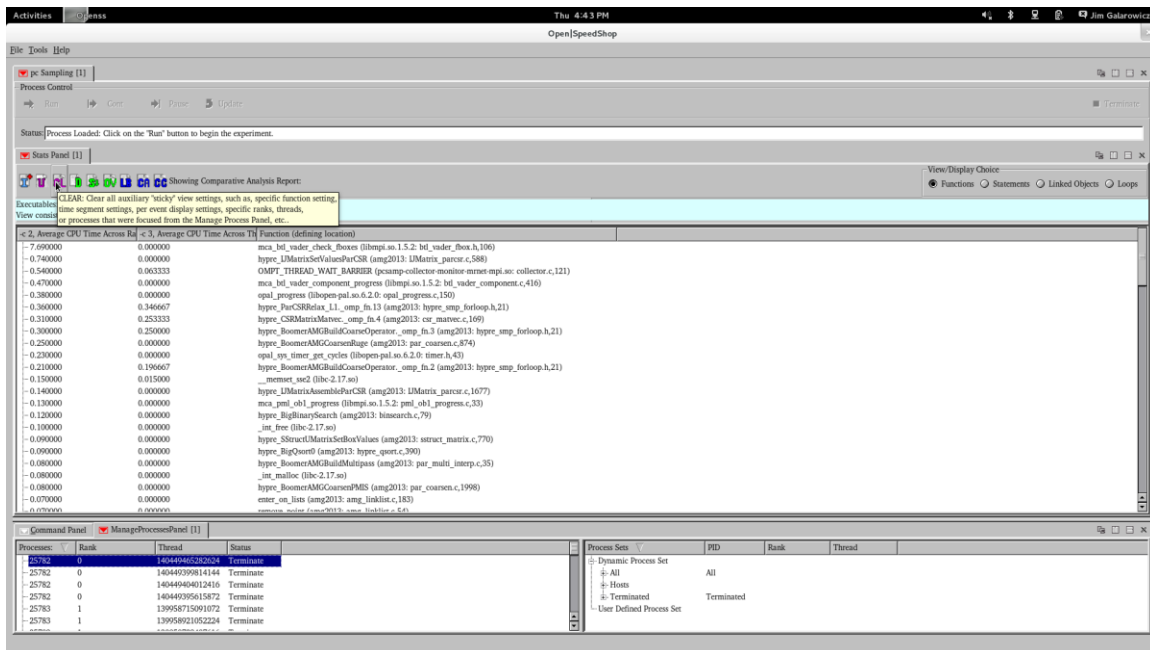
The next GUI view used the ManageProcess panel to highlight one rank, showing the performance data from all the threads executed under that particular rank in order to see only that performance data in the Stats Panel view.

Note: Use the "focus on selected rank and underlying threads" Manage Process panel option to focus on all the threads within a rank. Right mouse button down on the Manage Process panel tab to see the options.

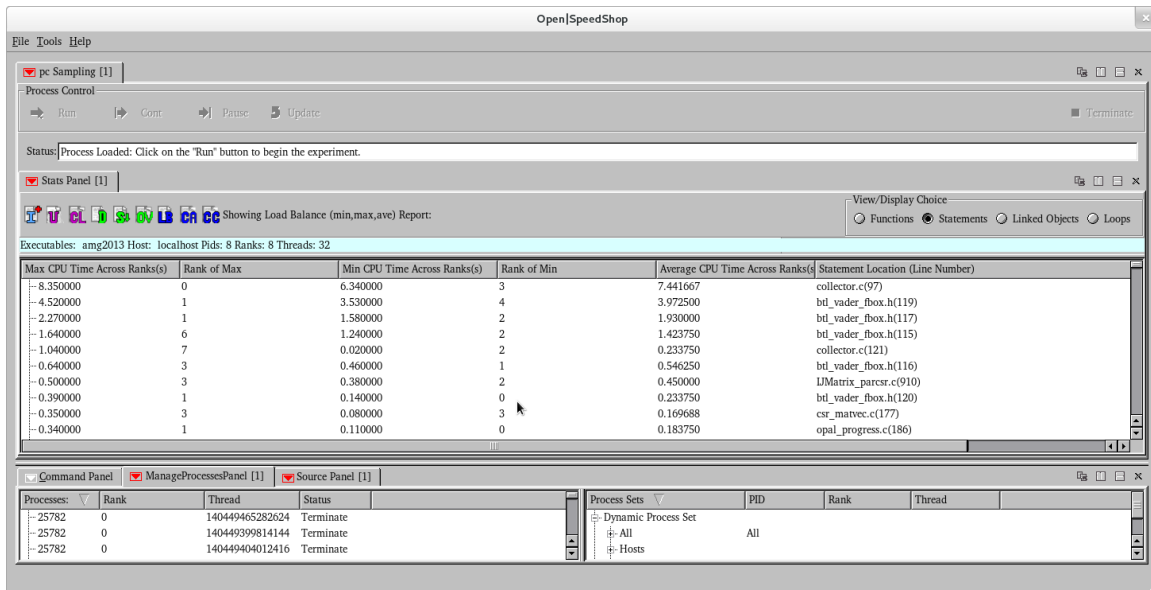


9.3.2 Clearing Focus on individual Rank to get back to default behavior

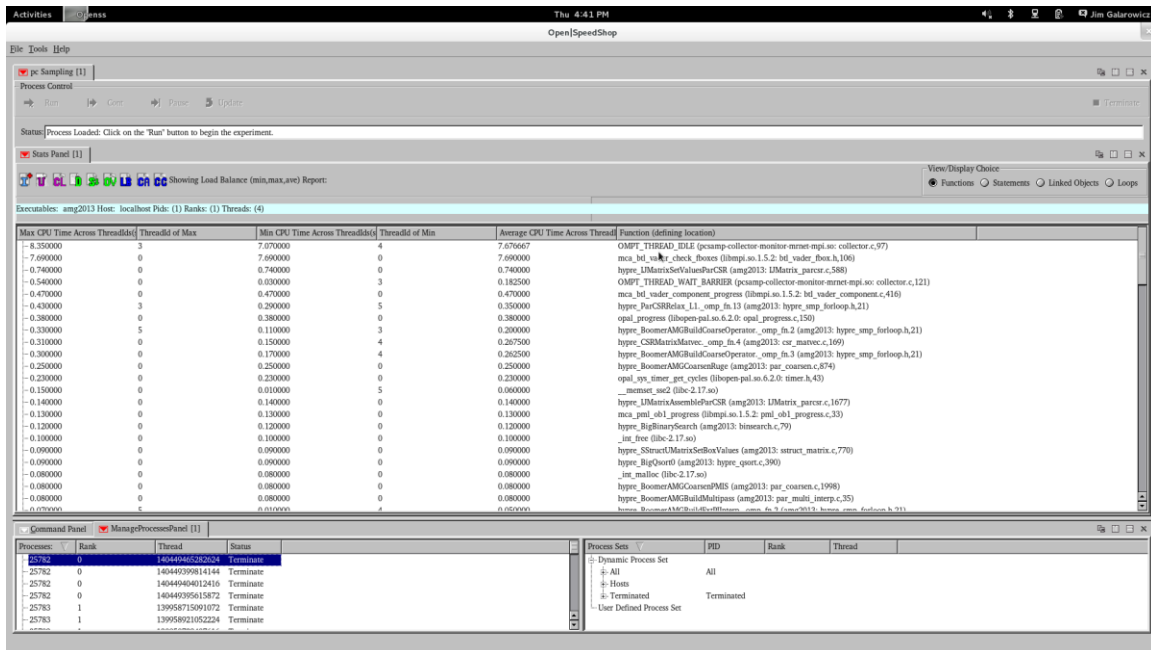
Note: Once the user focuses on individual or groups of ranks, e.g. venturing away from the default aggregated views, then the "CL" clear auxiliary setting icon is needed to clear all the optional selections and return to examining the aggregated results.



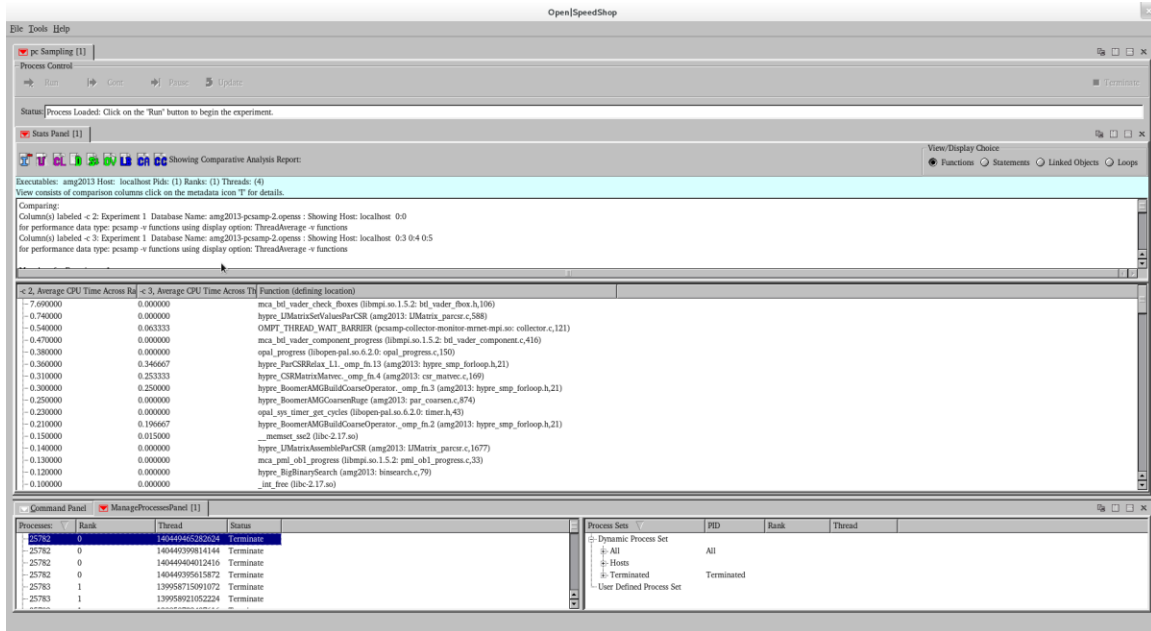
After clearing the specific rank and/or thread selections, clicking the "LB" load balance icon displays the min, max, average values across all ranks in the hybrid code. This helps decide if there is imbalance across ranks of the hybrid application. The user can focus on individual ranks to see the balance across the OpenMP threads that are in an individual rank:



This uses the Manage Process panel "Focus on selected rank and underlying threads" menu options to view the load balance across the four OpenMP threads for the rank 0 process:



The next GUI view selects the CA icon, which activates a cluster analysis algorithm on the performance data for the threads under rank 0. This view shows there are two groups of threads that are performing differently. Thread 0 is in one group and threads 3, 4, and 5 are in another group:



Please explore the options offered via a panel's pull-down menu. To access the options, click on a colored downward-facing arrow or use the Stats Panel icons. Red icons represent view options, such as updating the data or clearing the view options. The green icons correspond to possible performance data views. The dark blue icons correspond to analysis options while the light blue icon corresponds to information about the experiment. Hovering the cursor over the icons displays context-sensitive text.

10 GPU Performance Analysis

10.1 NVIDIA CUDA Analysis Section

The O|SS version with CBTF collection mechanisms supports tracing CUDA events in a NVIDIA CUDA based application. An event-by-event list of CUDA events and the event arguments are gathered and displayed.

10.1.1 NVIDIA CUDA Tracing (cuda) experiment performance data gathering (osscuda)

To run the NVIDIA CUDA experiment, use the osscuda convenience script and specify the CUDA application as an argument. Here is the general format of the osscuda convenience script that is used to gather the NVIDIA CUDA performance information.

```
osscuda "how you run your application normally"
```

In this example, the osscuda script will run the experiment by running the GEMM application and will create an OJSS database file with the results of the experiment. Viewing of the performance information can be done with the GUI or CLI. A default CLI text based report is displayed at the end of the application run.

```

osscuda "mpirun -np 2 -ppn 1 -hosts ccn001,ccn002 ./GEMM"
[openss]: cuda counting all instructions for CPU and GPU.
[openss]: cuda using default periodic sampling rate (10 ms).
[openss]: cuda configuration: "interval=10000000,PAPI_TOT_INS,inst_executed"
Creating topology file for slurm frontend node ccn001 for SLURM_JOB_ID 131
Generated topology file: ./cbtfAutoTopology
Running cuda collector.
Program: mpirun -np 2 -ppn 1 -hosts ccn001,ccn002 ./GEMM
Number of mrnet backends: 2
Topology file used: ./cbtfAutoTopology
executing mpi program: mpirun -np 2 -ppn 1 -hosts ccn001,ccn002 cbtfrun --mpi --mrnet -c
cuda ./GEMM
MPI Task 0/1 starting....
MPI Task 1/1 starting....
Chose device: name='Tesla K40c' index=0
Running single precision test
Chose device: name='Tesla K40c' index=0
Running single precision test
Running double precision test
Running double precision test
test atts units median mean stddev min max
DGEMM-N(max) 128 GFlops 57.5899 57.5899 0.259358 57.3306 57.8493
DGEMM-N(mean) 128 GFlops 56.9609 56.9609 0.0587321 56.9022 57.0196
DGEMM-N(median) 128 GFlops 57.3772 57.3772 0.395575 56.9816 57.7728
DGEMM-N(min) 128 GFlops 54.036 54.036 1.59142 52.4445 55.6274
...
...
Extreme outliers (>3.0 IQR from 1st/3rd quartile):
None.
default view for ./GEMM-cuda-3.openss

[openss]: The restored experiment identifier is: -x 1
Performance data spans 0.452563 ms from 2016/11/09 22:35:07 to 2016/11/09 22:35:07

Exclusive    % of Exclusive Function (defining location)
Time (ms)    Total    Count
Exclusive
Time
9.861216 52.062327    200 void RunTest<float>(std::string, ResultDatabase&, OptionParser&) (GEMM:
GEMM.cpp,156)
9.079958 47.937673    200 void RunTest<double>(std::string, ResultDatabase&, OptionParser&)
(GEMM: GEMM.cpp,156)

```

10.1.2 NVIDIA CUDA Experiment Performance Data Viewing using the new GUI

The NVIDIA CUDA Desktop application, having the executable name “opens-gui”, allows the user to explore CUDA event activity within in a timeline graph view and examine detailed parametric data for each CUDA event shown in the timeline view.

To launch the new beta, CUDA focused, GUI on any experiment, use:

```
“openss-gui -f <database name>“
```

This is a different GUI than the existing O|SS GUI. It is being developed initially focused on providing views for the NVIDIA CUDA experiment. Use openss-gui instead of openss to invoke this GUI. NOTE: However, this GUI will also load other non-CUDA experiments and if the collector type provides “time-based” metrics, such as “usertime”, then those are displayed.

10.1.3 NVIDIA CUDA GUI Main Window User Interface Layout

The application user interface is laid out in a logical manner to present a comprehensive view of the performance characteristics of an application. The main screen of the application is divided into four sections (ref. “*Figure 1 – Main Window User Interface Layout*”):

- Experiment Panel
- Metric Plot View
- Metric Table View
- Source Code View

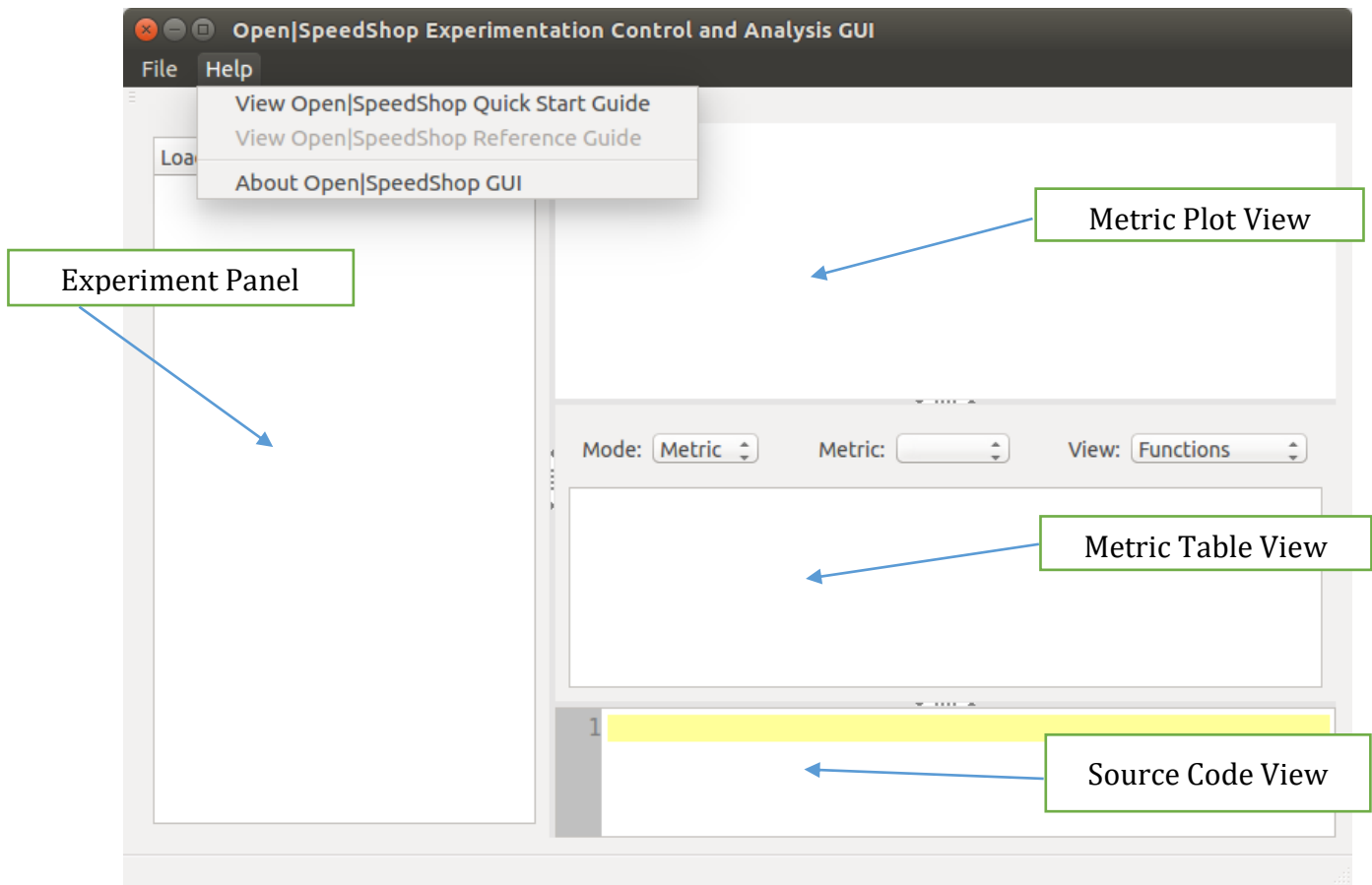


Figure 1- Main Window User Interface Layout

The Experiment Panel is on the left-hand side of the main window. Inside the Experiment Panel is the section labeled “Loaded Experiments” (ref. Figure 2). For the experiment that is currently loaded, this section shows the name of the loaded experiment (without the “.openss” file extension) at the top level of a tree view providing details regarding the application process, the CPU threads/ranks and GPU device. For CUDA experiments this information is shown under the tree view level titled “GPU Compute / Data Transfer Ratio”; otherwise this tree view level is titled “Thread Groups”. Currently, only one experiment can be loaded at a time. If another experiment is desired to be analyzed, then the user needs to unload the current experiment from the application before loading another. Experiment loading and unloading is accomplished using the menu items under the “File” menu. Each of the parallel executions (processes, threads, ranks, GPUs) are listed by hostname. The GPU device entry has “(GPU)” appended after the hostname. The checkbox to the left of the hostname is used to select which “thread groups” will be included in the performance views on the right-hand side of the main window (NOTE – currently all items are included regardless of selection). Under the thread/rank/GPU items is another subtree level enumerating which sample counters that were selected when the experiment was executed.

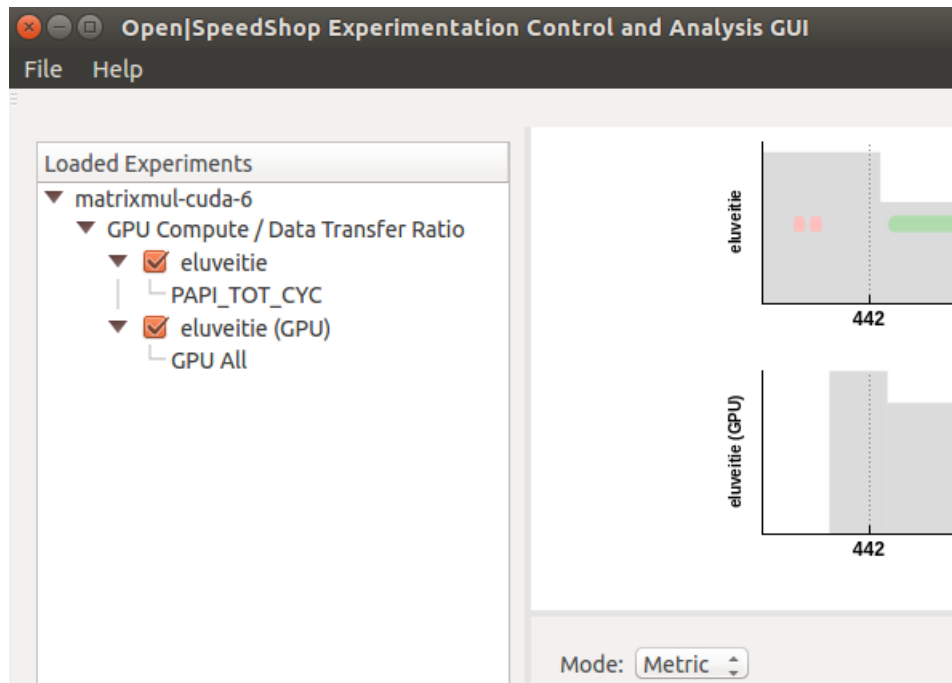


Figure 2- "Experiment Panel"

The right-hand side of the main window has the three other sections. The upper section is the Metric Plot View which provides a timeline in which the CUDA kernel executions are shown with a green color and data transfers between the GPU device and the CPU are shown with a red color. The length of the CUDA event in the timeline shows the duration of the event from the actual start (not enqueue time) and the end time of the event. The CUDA events are overlaid on top of a histogram showing the delta values as the counter is sampled by the collector (NOTE: currently only the first sample counter in the list can be viewed in the background histogram.) The CUDA event display provides insight into the relative cost of the transfers versus the actual time spent executing in the kernel.

The middle section on the right-hand side is the Metric Table View and is where the text based performance information is displayed in a table view. The user can control what type of information is displayed in the Metric Table View by using three different combo boxes. The "Mode" combo-box selects either "Metric" or "Details" view mode. Currently, the "Details" view mode only works for CUDA experiments and allows detailed examination of the CUDA events, filtered by type - Kernel Executions, Data Transfers or Both Kernel Executions and Data Transfers (All Events). The table view can be ordered by clicking on a column header to toggle between ascending and descending order using the selected column as the key for sorting. By default, the enqueue time column is sorted in ascending order. For the "Metric" view mode, the user will be able to view metric information, including: time, percentage, defining location, thread minimum, thread maximum and thread average. The metric type shown is selected using the "Metric" combo-box and the particular metric view can be changed with the

“View” combo-box. The “Metric” combo-box are metrics that can be selected in the OSS CLI using the “-m” option; whereas the “View” combo-box are views selectable in the OSS CLU using the “-v” option. For the “Metric” view the time interval for metric computations depends on the visible range of the graph timeline and for the “Detail” mode the same time interval is used to filter which CUDA events are shown in the table. As the user changes the graph timeline by zooming into the graph or panning the timeline left or right the Metric Table View is dynamically updated. There is a delay threshold between the time the user pauses or completes timeline changes and the actual kickoff of the processing involved for the Metric Plot or Metric Table View updates.

The lower section on the right-hand side is the Source Code View. When the user has activated the “Metric” mode of the Metric Table View, any selections of a row in the table cause the display the corresponding line of the source code in the Source Code View. Updates to the Source Code View is possible in either the “Functions”, “Statements” or “Loops” metric view (but not the “Linked Objects” metric view) as long as the source code is available on the host machine. If the source code is not in the same location as when the executable was compiled, then the user can specify the mapping between the original development machine location and the location of the host machine processing the experiment database (running the opens-gui application). The dialog in which the mappings can be specified is by activating a context menu. The context menu is activated by holding down the right-mouse button when the cursor is over the table row of interest. When the context menu appears near the location of the cursor, the user must select the “Modify Path Substitutions” menu item to activate the “Modify Path Substitutions Dialog” (ref “Figure 3 - Modify Path Substitutions Dialog”) . The “Modify Path Substitutions Dialog” shows a table with two columns – the left column are the original paths to the source code and the right column are the paths on the local host machine. When the dialog is activated a new entry in the table is created with the left column, “Original Path”, filled in from the information in the metric data. The user then types in the absolute path on the local host machine to the corresponding source code.

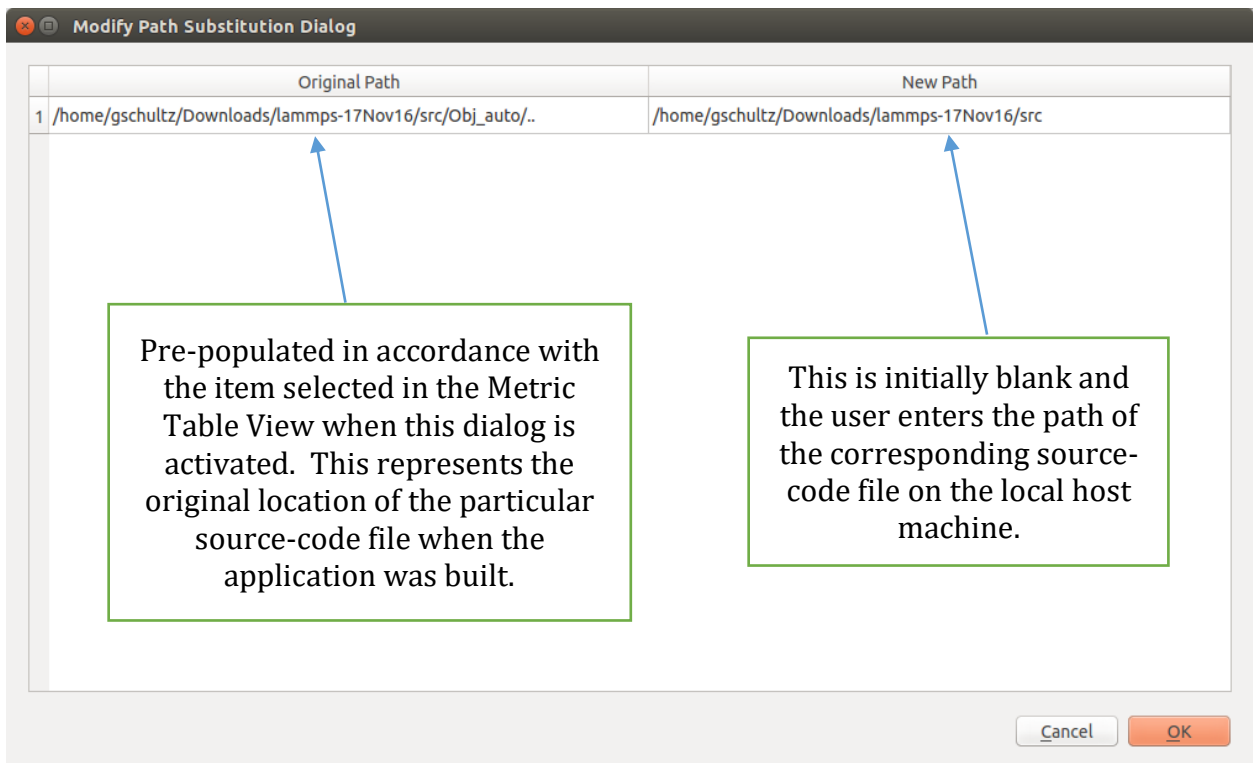


Figure 3 - Modify Path Substitutions Dialog

10.1.4 Using the NVIDIA CUDA GUI to Analyze Application Performance

In order to demonstrate how the new GUI can be used to view CPU and GPU activity within an application and generate summary metric results and detailed CUDA event lists two different examples will be discussed.

To launch the new GUI using the GEMM experiment discussed in the previous section, use:

```
openss-gui -f GEMM-cuda-0.openss
```

The default view for the new GUI using the GEMM experiment discussed previously can be seen in Figure 4. As seen here the main window configuration was changed by the user to completely close the “Experiment Panel” normally visible on the left-hand side of the main window so that the right-hand panels take the full width of the main window. This is accomplished by using the “handles” in the border area between two panels (ref. the annotation in Figure 4 and Figure 5 for a zoomed in view of the splitter handle between the Metric Plot and Metric Table Views).

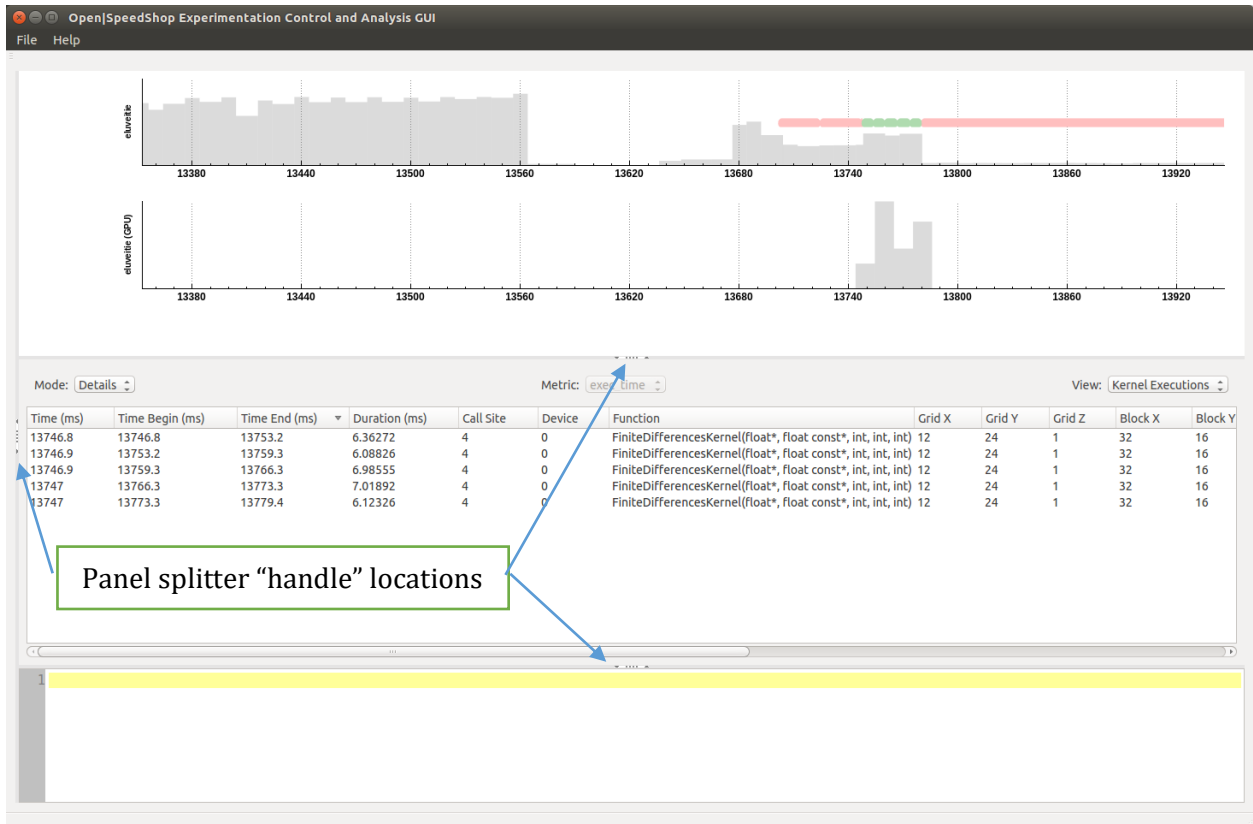


Figure 4- Default View for the GEMM Experiment

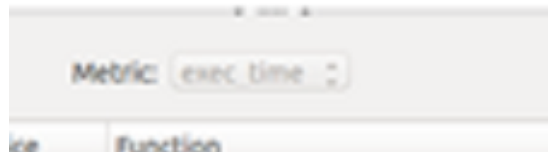


Figure 5- Zoomed View of Panel Splitter Handles

For the screenshot shown in Figure 6 one can see the CUDA events in the graph timeline. The CUDA events are currently placed on the CPU graph of the CPU + GPU graph view. The current thought process for placing them on the CPU graph is so that it does not obstruct the GPU sample counter histogram and the user can clearly see the magnitude of each histogram bar as there should be a direct relationship with CUDA event activity. As discussed previously the red pastel colored rectangle corresponds to a Data Transfer event and the green pastel colored rectangle to a Kernel Execution event. Thus, in the graph above there are two Data Transfer events, followed by 5 Kernel Execution events, followed by one Data Transfer event (see annotations on screenshot). There is another annotation from the "Time Begin (ms)" value of the first Kernel Execution to the position of the left-edge of the Kernel Execution event rectangle on the graph timeline. The "Time End (ms)" value will be the position on the graph timeline for the right

edge of the Kernel Execution event rectangle. This screenshot represents the “Details – All Events” view in the area below the Metric Plot View. The additional two screenshots show the “Details – Data Transfers” and “Details – Kernel Executions” views that just contain CUDA Data Transfer or CUDA Kernel Execution events respectively (ref. Figures 7 and 8).

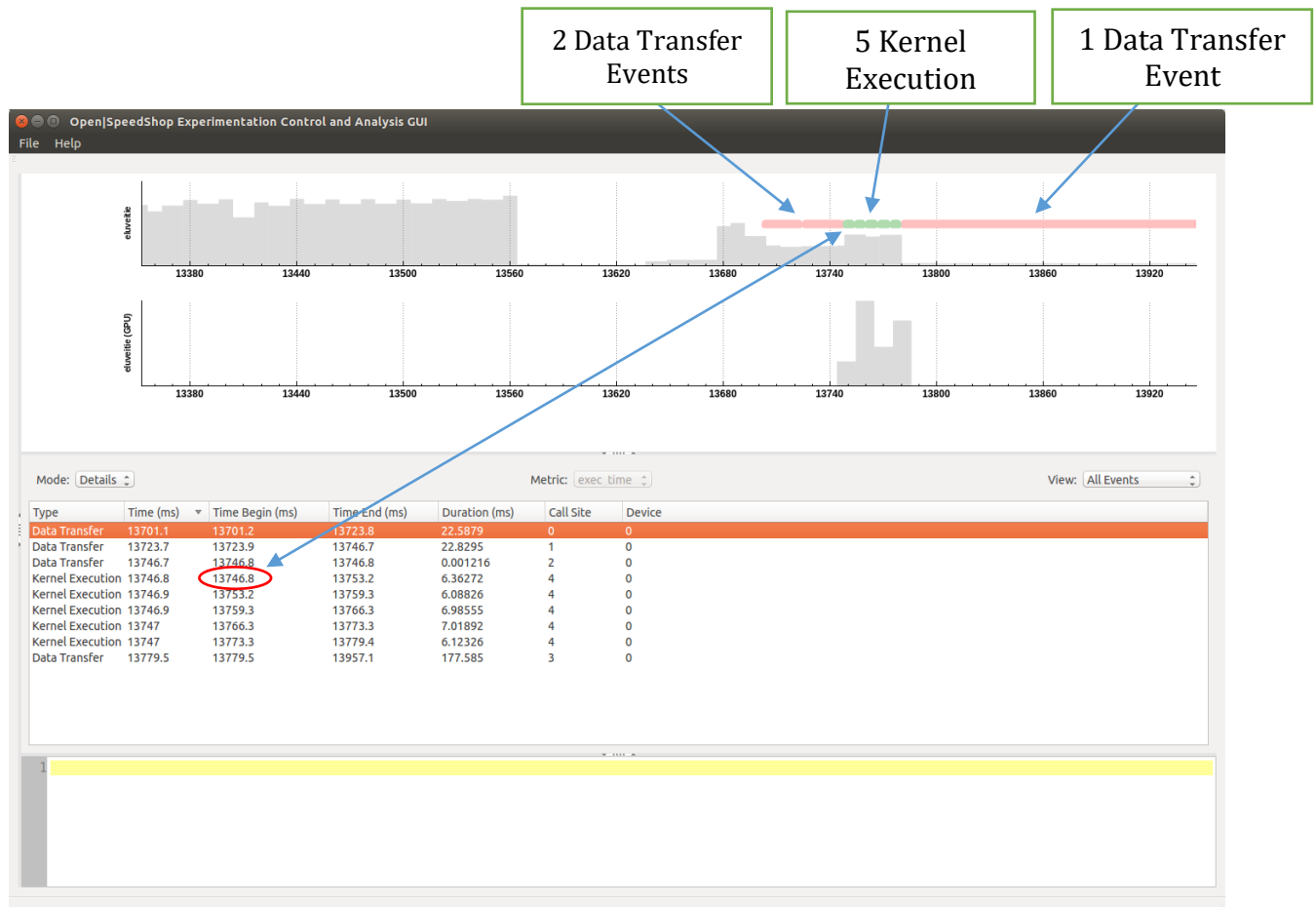


Figure 6 - CUDA Events in Graph Timeline and Details Mode View

For the Data Transfer and Kernel Execution Details views many more columns are displayed showing all the available event information. For the All Events Details view only the common set of event information is shown.

As discussed previously the metric values displayed in the “Metric” mode or the events listed in the various “Details” mode views use the visible time range in the graph timeline as input to the metric computations or filtering logic for which CUDA events to show.

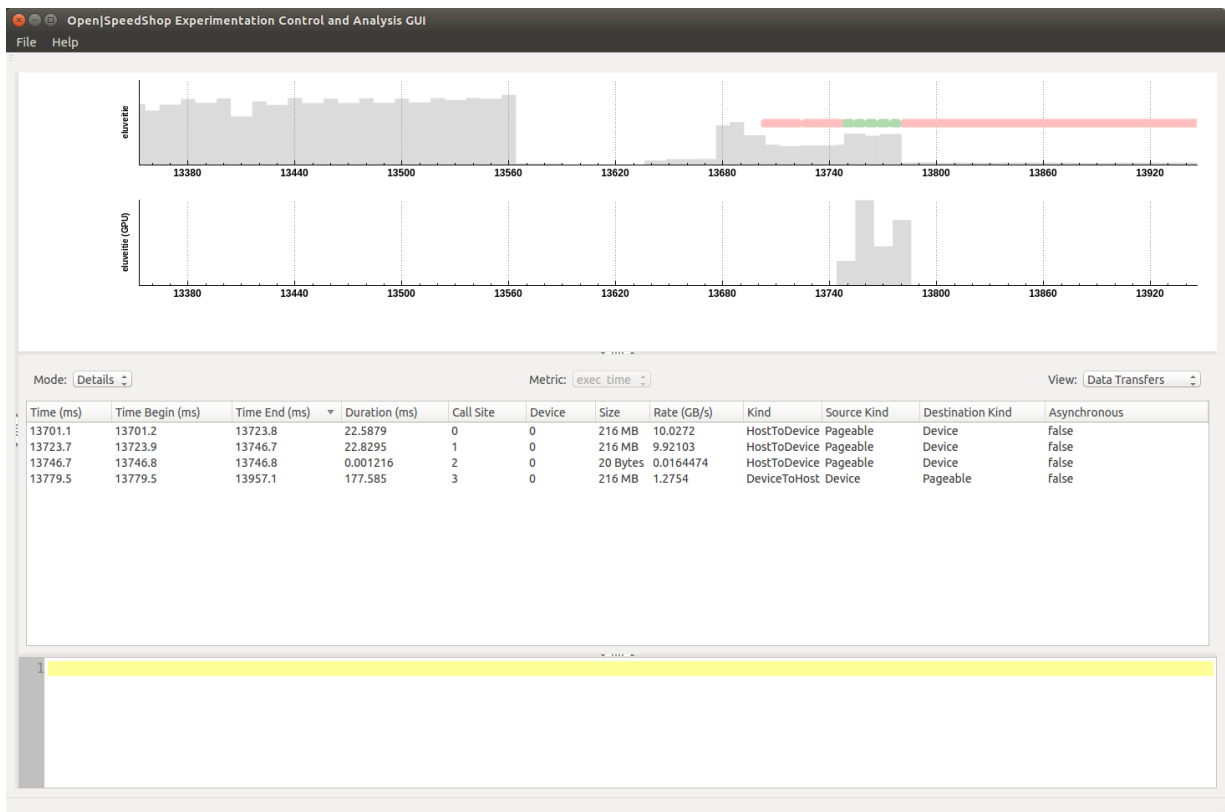


Figure 7- Data Transfer Details View

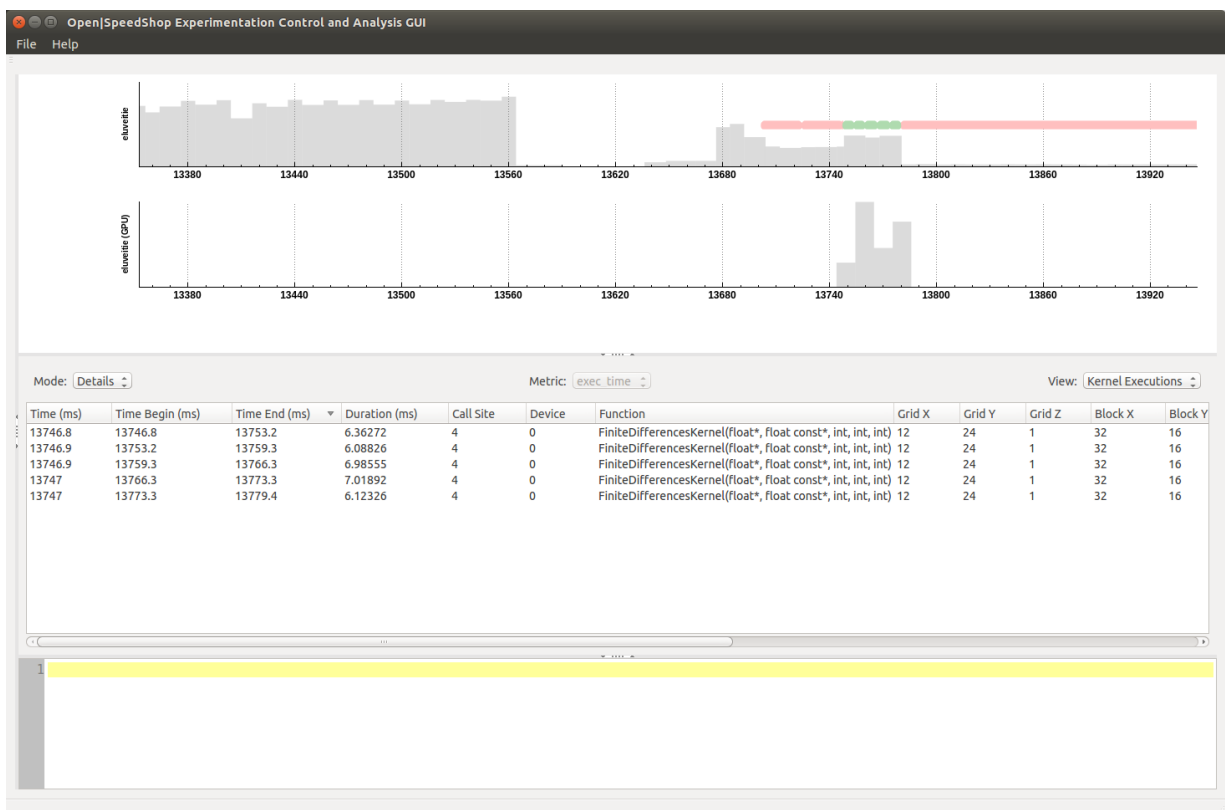


Figure 8- Kernel Execution Details View

Let's show another CUDA experiment starting with the performance data collection by running the "osscuda" convenience script on an example CUDA program which executes various implementations of matrix multiplication to demonstrate performance differences using various performance optimization techniques, including:

1. Tiling
2. Memory coalescing
3. Avoiding memory bank conflicts
4. Increase floating portion by outer product.
5. Loop unrolling
6. Prefetching

A discussion of the matrix multiplication problem, the various performance optimization techniques used in the application and source-code can be found at <https://sites.google.com/site/5kk70gpu/matrixmul-example>.

```
$ osscuda "./matrixmul"
[openss]: cuda counting all instructions for CPU and GPU.
[openss]: cuda using default periodic sampling rate (10 ms).
[openss]: cuda configuration: "interval=10000000,PAPI_TOT_INS,inst_executed"
Creating topology file for frontend host eluv
Generated topology file: ./cbtfAutoTopology
Running cuda collector.
Program: ./matrixmul
Number of mrnet backends: 1
Topology file used: ./cbtfAutoTopology
executing sequential program: cbtfrun -c cuda --mrnet ./matrixmul
[Matrix Multiply Using CUDA] - Starting...
GPU Device 0: "GeForce GTX 1060" with compute capability 6.1

[CUDA 5632:0] CUPTI_metrics_start(): The selected CUDA device doesn't support continuous GPU event
sampling. GPU events will be sampled at CUDA kernel entry and exit only (not periodically). This also
implies CUDA kernel execution will be serialized, possibly exhibiting different temporal behavior than
when executed without performance monitoring.
Naive CPU (Golden Reference)
Processing time: 279.404175 (ms), GFLOPS: 0.360278
threads: x=16 y=16
grid: x=24 y=16
Naive GPU
Processing time: 1.555232 (ms), GFLOPS: 64.725580
Total Errors = 0
Tiling GPU
Processing time: 0.944896 (ms), GFLOPS: 106.533736
Total Errors = 0
Global mem coalescing GPU
Processing time: 1.168640 (ms), GFLOPS: 86.137128
Total Errors = 0
```

```

Remove shared mem bank conflict GPU
Processing time: 0.853728 (ms), GFLOPS: 117.910264
Total Errors = 0
Threads perform computation optimization GPU
Processing time: 0.825312 (ms), GFLOPS: 121.969984
Total Errors = 0
Loop unrolling GPU
Processing time: 0.862624 (ms), GFLOPS: 116.694296
Total Errors = 0
Prefetching GPU
Processing time: 1.037664 (ms), GFLOPS: 97.009520
Total Errors = 0
default view for /home/gschultz/Downloads/exercises/cuda/matrixMul/matrixmul-cuda-3.openss
[openss]: The restored experiment identifier is: -x 1
Performance data spans 0.461198 ms from 2017/02/16 23:26:30 to 2017/02/16 23:26:31

Exclusive   % of Exclusive Function (defining location)
Time (ms)   Total    Count
    Exclusive
    Time
0.605867 32.275192      1 matrixMul_coalescing(float*, float*, float*, int, int) (matrixmul:
matrixMul_coalescing.cuh,31)
0.496201 26.433165      1 matrixMul_naive(float*, float*, float*, int, int) (matrixmul:
matrixMul_naive.cuh,17)
0.257925 13.739944      1 matrixMul_tiling(float*, float*, float*, int, int) (matrixmul:
matrixMul_tiling.cuh,31)
0.211493 11.266461      1 matrixMul_noBankConflict(float*, float*, float*, int, int) (matrixmul:
matrixMul_noBankConflict.cuh,32)
0.108675  5.789235      1 matrixMul_prefetch(float*, float*, float*, int, int) (matrixmul:
matrixMul_prefetch.cuh,31)
0.107011  5.700592      1 matrixMul_compOpt(float*, float*, float*, int, int) (matrixmul:
matrixMul_compOpt.cuh,31)
0.090019  4.795410      1 matrixMul_unroll(float*, float*, float*, int, int) (matrixmul:
matrixMul_unroll.cuh,32)

```

Upon completion of the CUDA experiment the O|SS experiment database can be located. For this run it is in the file “matrixmul-cuda-3.openss”. First let’s open the experiment in the O|SS CLI:

```
opens -cli -f matrixmul-cuda-3.openss
```

Once the CLI has loaded the experiment the following series of commands are generated to produce metric data:

```

expview -vexec -mexclusive_time,threadmin,threadmax,avg -l432.892:444.104
expview -vxfer -mexclusive_time,threadmin,threadmax,avg -l432.892:444.104
expview -vexec -mexclusive_time,threadmin,threadmax,avg -l441.384:443.981

```

```
expview -vxfer -mexclusive_time,threadmin,threadmax,avg -I441.384:443.981
```

```

openss>>expview -vexec -mexclusive_time,threadmin,threadmax,avg -I432.892:444.104

Exclusive      Min CUDA      Max CUDA      Average      Function (defining location)
Time (ms)      Kernel        Kernel        Time
                Execution      Execution      (ms)
                Time Across   Time Across
                ThreadIds (ms) ThreadIds (ms)

0.605867      0.605867      0.605867      0.605867      matrixMul_coalescing(float*, float*,
float*, int, int) (matrixmul: matrixMul_coalescing.cuh,31)
0.496201      0.496201      0.496201      0.496201      matrixMul_naive(float*, float*, float*,
int, int) (matrixmul: matrixMul_naive.cuh,17)
0.257925      0.257925      0.257925      0.257925      matrixMul_tiling(float*, float*, float*,
int, int) (matrixmul: matrixMul_tiling.cuh,31)
0.211493      0.211493      0.211493      0.211493      matrixMul_noBankConflict(float*, float*,
float*, int, int) (matrixmul: matrixMul_noBankConflict.cuh,32)
0.108675      0.108675      0.108675      0.108675      matrixMul_prefetch(float*, float*,
float*, int, int) (matrixmul: matrixMul_prefetch.cuh,31)
0.107011      0.107011      0.107011      0.107011      matrixMul_compOpt(float*, float*, float*,
int, int) (matrixmul: matrixMul_compOpt.cuh,31)
0.090019      0.090019      0.090019      0.090019      matrixMul_unroll(float*, float*, float*,
int, int) (matrixmul: matrixMul_unroll.cuh,32)
openss>>expview -vxfer -mexclusive_time,threadmin,threadmax,avg -I432.892:444.104

Exclusive      Min CUDA      Max CUDA      Average      Function (defining location)
Time (ms)      Data          Data          Time
                Transfer      Transfer      (ms)
                Time Across   Time Across
                ThreadIds (ms) ThreadIds (ms)

0.973283      0.973283      0.973283      0.046347      runTest(int, char**) (matrixmul:
matrixMul.cu,163)
openss>>expview -vexec -mexclusive_time,threadmin,threadmax,avg -I441.384:443.981

Exclusive      Min CUDA      Max CUDA      Average      Function (defining location)
Time (ms)      Kernel        Kernel        Time
                Execution      Execution      (ms)
                Time Across   Time Across
                ThreadIds (ms) ThreadIds (ms)

0.108675      0.108675      0.108675      0.108675      matrixMul_prefetch(float*, float*,
float*, int, int) (matrixmul: matrixMul_prefetch.cuh,31)
0.090019      0.090019      0.090019      0.090019      matrixMul_unroll(float*, float*, float*,
int, int) (matrixmul: matrixMul_unroll.cuh,32)
openss>>expview -vxfer -mexclusive_time,threadmin,threadmax,avg -I441.384:443.981

Exclusive      Min CUDA      Max CUDA      Average      Function (defining location)
Time (ms)      Data          Data          Time
                Transfer      Transfer      (ms)
                Time Across   Time Across
                ThreadIds (ms) ThreadIds (ms)

0.287658      0.287658      0.287658      0.047943      runTest(int, char**) (matrixmul:
matrixMul.cu,163)

```

Now let's launch the new GUI automatically loading the same experiment database:

```
openss-gui -f matrixmul-cuda-3.openss
```

The screenshot shows the OpenSpeedShop Experimentation Control and Analysis GUI. The top panel displays a timeline of events (red dots) and GPU usage (gray bars) across iterations 434 to 444. The bottom panel shows a table of metrics for different CUDA kernel execution times, with the 'matrixMul_tiling' function highlighted in orange.

CUDA Kernel Execution Time (msec)	Minimum (Thread)	Maximum (Thread)	Average (Thread)	Function (defining location)
0.605867	0.605867	0.605867	0.605867	matrixMul_coalescing(float*, float*, float*, int, int) (/home/gschultz/Downloads/exercises/cuda/matrixMul/./matrixMul_coalescing.cu)
0.496201	0.496201	0.496201	0.496201	matrixMul_naive(float*, float*, float*, int, int) (/home/gschultz/Downloads/exercises/cuda/matrixMul/./matrixMul_naive.cu)
0.257925	0.257925	0.257925	0.257925	matrixMul_tiling(float*, float*, float*, int, int) (/home/gschultz/Downloads/exercises/cuda/matrixMul/./matrixMul_tiling.cu)
0.211493	0.211493	0.211493	0.211493	matrixMul_noBankConflict(float*, float*, float*, int, int) (/home/gschultz/Downloads/exercises/cuda/matrixMul/./matrixMul_noBankConflict.cu)
0.108675	0.108675	0.108675	0.108675	matrixMul_prefetch(float*, float*, float*, int, int) (/home/gschultz/Downloads/exercises/cuda/matrixMul/./matrixMul_prefetch.cu)
0.107011	0.107011	0.107011	0.107011	matrixMul_compOpt(float*, float*, float*, int, int) (/home/gschultz/Downloads/exercises/cuda/matrixMul/./matrixMul_compOpt.cu)
0.090019	0.090019	0.090019	0.090019	matrixMul_unroll(float*, float*, float*, int, int) (/home/gschultz/Downloads/exercises/cuda/matrixMul/./matrixMul_unroll.cu)

132

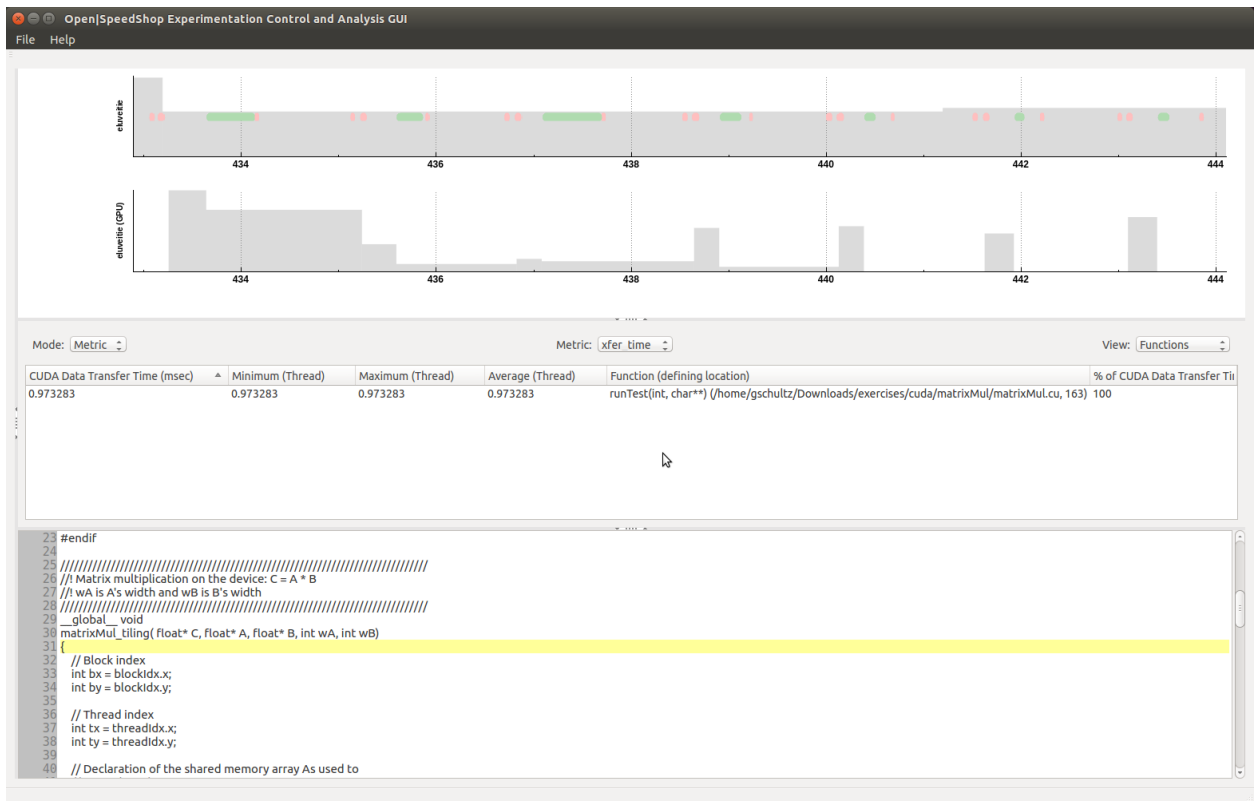


Figure 10- "expview -vxfer -mexclusive_time,threadmin,threadmax,avg -l432.892:444.104"

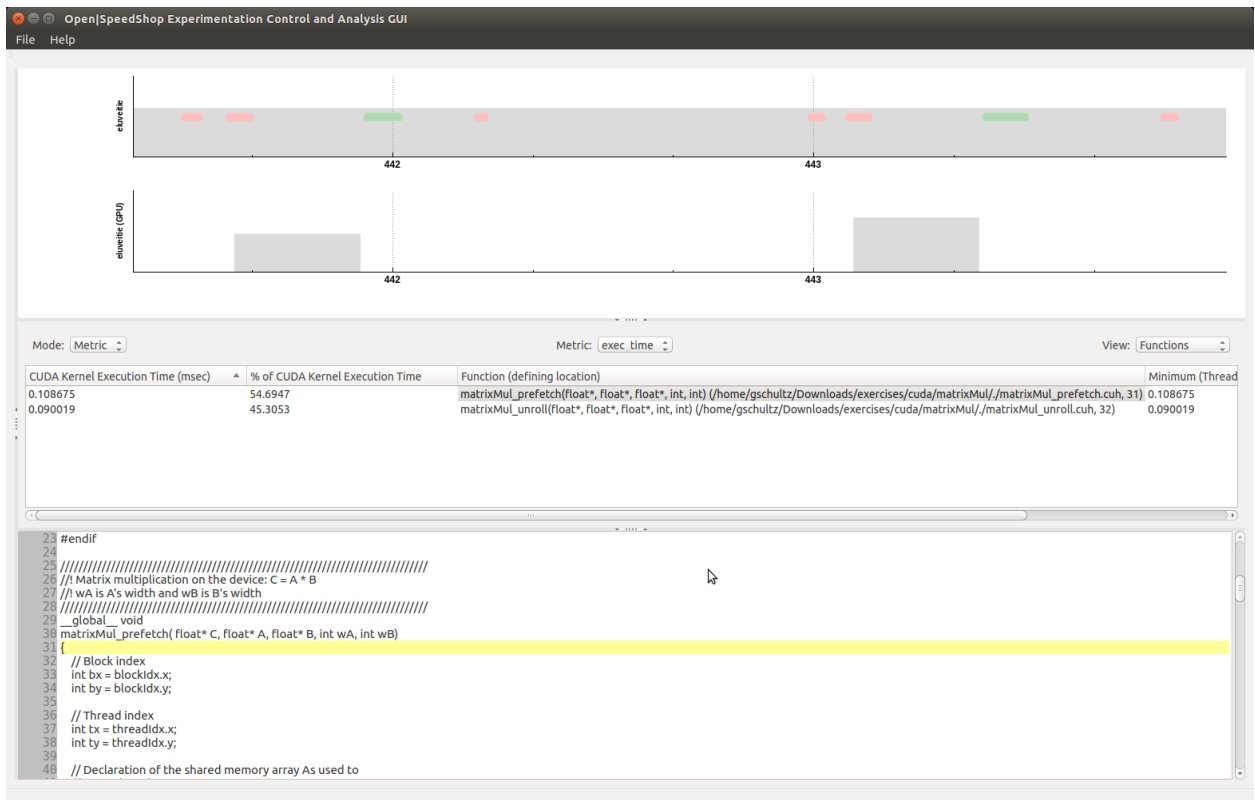


Figure 11- "expview -vexec -mexclusive_time,threadmin,threadmax,avg -l441.384:443.981"

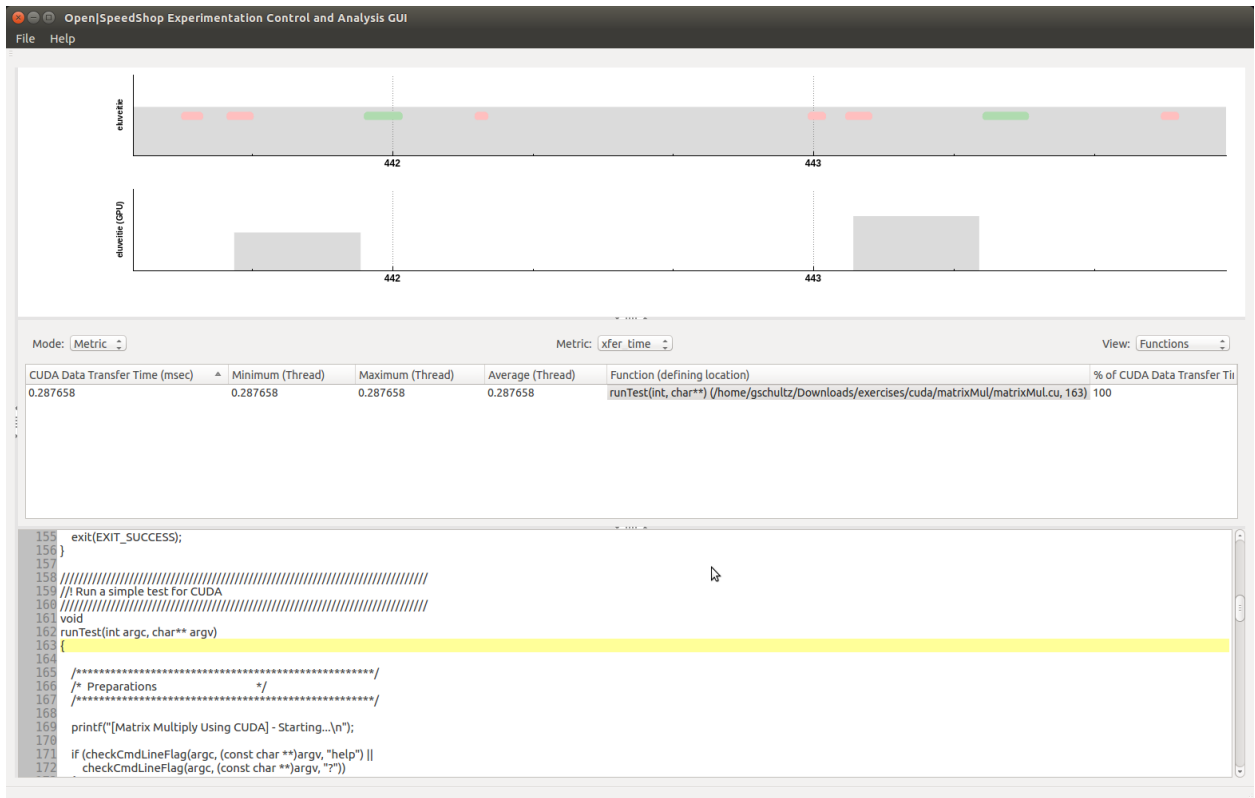


Figure 12- "expview -vxfer -mexclusive_time,threadmin,threadmax,avg -l441.384:443.981"

Each screenshot caption indicates the corresponding “expview” command in the O|SS CLI.

These screenshots demonstrate that the user can alter the column ordering by holding the left-mouse button when the mouse cursor is over one of the columns and moving it into a new position. The columns were re-ordered to match the ordering of the CLI views.

10.1.5 Viewing NVIDIA CUDA Tracing (cuda) experiment performance data via CLI

To launch the CLI on any experiment, use “openss -cli -f <database name>”.

The O|SS CLI will report NVIDIA CUDA kernel execution, NVIDIA CUDA data transfer and CPU/GPU hardware performance counter data the “cuda” collector gathers.

The type of data displayed can be controlled through the '-v' options:

Exec	CUDA kernel executions (this is the default)
Xfer	CUDA data transfers

The form of the displayed information is controlled thru additional '-v' options. For '-v Exec' and '-v Xfer' these additional '-v' options are:

ButterFly	Produces a report summarizing the calls to and from the one or more functions specified by the '-f <function_list>' option. By default, calling functions will be listed before the named function and called functions afterwards, unless 'TraceBacks' is specified to reverse this order.
CallTree[s]	Produces a calling stack report presented in calling tree order, from the executable's start toward the measurement locations.
(DSO LinkedObject)[s]	Produces a summary report by linked object.
FullStack[s]	Causes the report to include the full call stack for each measurement location when added to either 'CallTree' or 'TraceBack'. Redundant call stack frames are suppressed by default if this option isn't specified.
Function[s]	Produces a summary report by function. This is the default.
Loop[s]	Produces a summary report by loop.
Statement[s]	Produces a summary report by statement.
Summary	Causes the report to include an additional line of output at the end that summarizes the information in each column. Does not apply to 'ButterFly' or 'Trace'.
SummaryOnly	Causes the report to ONLY include the line of output generated by 'Summary'.
Trace	Produces a report of each individual CUDA kernel execution or data transfer, sorted in ascending order of the event's start time.
TraceBack[s]	Produces a calling stack report presented in traceback order, from the measurement

locations toward the executable's start.

Except for the '-v Trace' option, the report will be sorted in descending order of values in the leftmost column. Multiple '-v' values can be delimited with commas, e.g. '-v Exec,Trace'.

Finally, columns included in the report can be controlled using the '-m' option. More than one column may be specified in a comma-delimited list. And when '-m' is used, ONLY those columns specified are reported, in the order given.

The following '-m' options are available for '-v [Exec|Xfer]':

[%][exclusive_]count[s]	Exclusive number of events
[%][inclusive_]count[s]	Inclusive number of events
[%][exclusive_]time[s]	Exclusive time in the event
[%][inclusive_]time[s]	Inclusive time in the event
min[imum]	Minimum time in the event
max[imum]	Maximum time in the event
avg[erage]	Average time in the event
stddev	Standard deviation of time in the event
ThreadMin	Minimum accumulated time for a process
ThreadMinIndex	Process ID of the 'ThreadMin' process
ThreadMax	Maximum accumulated time for a process
ThreadMaxIndex	Process ID of the 'ThreadMax process'
ThreadAverage	Average accumulated time for a process
LoadBalance	Equivalent to 'ThreadMax, ThreadMaxIndex, ThreadMin, ThreadMinIndex, ThreadAverage'.

The following '-m' options are only available for '-v [Exec|Xfer],Trace':

(start|stop)[_time] Start or stop time for the event

The following '-m' options are only available for '-v Exec,Trace':

block	Dimensions of each block
cache	Cache preference used
dsm	Total amount (in bytes) of dynamic shared memory reserved
grid	Dimensions of the grid
lm	Total amount (in bytes) of local memory reserved
rpt	Registers required for each thread
ssm	Total amount (in bytes) of static shared reserved

The following '-m' options are only available for '-v Xfer,Trace':

size	Number of bytes being transferred
kind	Kind of data transfer performed
src	Kind of memory from which the data transfer was performed
dest	Kind of memory to which the data transfer was performed
async	Was the data transfer asynchronous?

The default columns used for various '-v' combinations are:

-v Exec,Trace	-m start,time,%time,grid,block
-v Xfer,Trace	-m start,time,%time,size,kind
-v (Exec Xfer),Butterfly	-m inclusive_time,%inclusive_time
-v (Exec Xfer)[,<all-other>]	-m time,%time,count

The '-v HWPC' view works differently: It only displays the sampled CPU/GPU hardware performance counters as a function of time; i.e. it does not display data as a function of source-code constructs. Thus, only the '-v Summary' and '-v SummaryOnly' options apply.

It also interprets differently the positive integer added to the end of the keyword 'cuda'. Instead of identifying the maximum number of reported items, it specifies the fixed sampling interval (in ms) at which the data should be resampled before display. The default value (0) is given the special meaning that the original sampling interval should be used instead.

Examples:

```
expView cuda
expView -v Xfer,Fullstack cuda10 -m min,max,count
expView -v HWPC,Summary cuda33
```

See also:

expView

Here are some CLI views of the output from the osscuda experiment. These views show results of a cuda experiment on the NVIDIA CUDA application GEMM on the Pleiades SGI platform at NASA:

```
pfe27-433>openss -cli -f GEMM-cuda-4.openss
openss>>[openss]: The restored experiment identifier is: -x 1
openss>>expview

Exclusive   % of Exclusive  Function (defining location)
Time (ms)   Total    Count
    Exclusive
    Time
14.810702  52.042113   300 void RunTest<float>(std::string, ResultDatabase&, OptionParser&) (GEMM: GEMM.cpp,19)
13.648369  47.957887   300 void RunTest<double>(std::string, ResultDatabase&, OptionParser&) (GEMM: GEMM.cpp,19)
openss>>expstatus
```

```

Experiment definition
{ # ExpId is 1, Status is Terminated, Saved database is GEMM-cuda-4.openss
  Performance data spans 0.443760 ms from 2016/08/24 10:01:03 to 2016/08/24 10:01:03
  (none)
  Executables Involved:
    (none)
  Currently Specified Components:
    -h maia29 -p 43727 -t 0 -r 0
    -h maia30 -p 27136 -t 0 -r 1
    -h maia31 -p 80595 -t 0 -r 2
  Previously Used Data Collectors:
    cuda
  Metrics:
    cuda::count_exclusive_details
    cuda::exec_exclusive_details
    cuda::exec_inclusive_details
    cuda::exec_time
    cuda::xfer_exclusive_details
    cuda::xfer_inclusive_details
    cuda::xfer_time
  Parameter Values:
  Available Views:
    cuda
}

openss>>expview -vExec

Exclusive   % of Exclusive Function (defining location)
Time (ms)   Total   Count
  Exclusive
    Time
14.810702 52.042113   300 void RunTest<float>(std::string, ResultDatabase&, OptionParser&) (GEMM:
GEMM.cpp,19)
13.648369 47.957887   300 void RunTest<double>(std::string, ResultDatabase&, OptionParser&) (GEMM:
GEMM.cpp,19)
openss>>expview -vXfer

Exclusive   % of Exclusive Function (defining location)
Time (ms)   Total   Count
  Exclusive
    Time
1.774178 75.232917   69 void RunTest<float>(std::string, ResultDatabase&, OptionParser&) (GEMM:
GEMM.cpp,19)
0.584069 24.767083   69 void RunTest<double>(std::string, ResultDatabase&, OptionParser&) (GEMM:
GEMM.cpp,19)
openss>>expview -v trace, Xfer
  Start Time (d:h:m:s)   Exclusive   % of   Size   Kind Call Stack Function (defining location)
                        Time (ms)   Total
                        Exclusive
                        Time
2016/08/24 10:01:03.845 0.001217 0.051606 112 HostToDevice >>void RunTest<float>(std::string,
ResultDatabase&, OptionParser&) (GEMM: GEMM.cpp,19)
2016/08/24 10:01:03.850 0.027392 1.161541 262144 HostToDevice >>void RunTest<float>(std::string,
ResultDatabase&, OptionParser&) (GEMM: GEMM.cpp,19)
2016/08/24 10:01:03.850 0.027553 1.168368 262144 HostToDevice >>void RunTest<float>(std::string,
ResultDatabase&, OptionParser&) (GEMM: GEMM.cpp,19)
2016/08/24 10:01:03.851 0.001217 0.051606 112 HostToDevice >>void RunTest<float>(std::string,
ResultDatabase&, OptionParser&) (GEMM: GEMM.cpp,19)
2016/08/24 10:01:03.851 0.027425 1.162940 262144 DeviceToHost >>void RunTest<float>(std::string,
ResultDatabase&, OptionParser&) (GEMM: GEMM.cpp,19)
2016/08/24 10:01:03.852 0.026721 1.133087 262144 DeviceToHost >>void RunTest<float>(std::string,
ResultDatabase&, OptionParser&) (GEMM: GEMM.cpp,19)
2016/08/24 10:01:03.852 0.026753 1.134444 262144 DeviceToHost >>void RunTest<float>(std::string,
ResultDatabase&, OptionParser&) (GEMM: GEMM.cpp,19)
.....

```

```
openss>>expview -v trace,Exec
```

Start Time (d:h:m:s)	Exclusive Time (ms)	% of Total Exclusive Time	Grid Dims	Block Dims	Call Stack Function (defining location)
2016/08/24 10:01:03.851	0.055585	0.195316	4,4,1	16,16,1	>>void RunTest<float>(std::string, ResultDatabase&, OptionParser&) (GEMM: GEMM.cpp,19)
2016/08/24 10:01:03.851	0.048705	0.171141	4,4,1	16,16,1	>>void RunTest<float>(std::string, ResultDatabase&, OptionParser&) (GEMM: GEMM.cpp,19)
2016/08/24 10:01:03.851	0.049761	0.174851	4,4,1	16,16,1	>>void RunTest<float>(std::string, ResultDatabase&, OptionParser&) (GEMM: GEMM.cpp,19)
2016/08/24 10:01:03.851	0.051617	0.181373	4,4,1	16,16,1	>>void RunTest<float>(std::string, ResultDatabase&, OptionParser&) (GEMM: GEMM.cpp,19)
2016/08/24 10:01:03.851	0.051648	0.181482	4,4,1	16,16,1	>>void RunTest<float>(std::string, ResultDatabase&, OptionParser&) (GEMM: GEMM.cpp,19)
2016/08/24 10:01:03.851	0.050817	0.178562	4,4,1	16,16,1	>>void RunTest<float>(std::string, ResultDatabase&, OptionParser&) (GEMM: GEMM.cpp,19)
2016/08/24 10:01:03.851	0.046496	0.163378	4,4,1	16,16,1	>>void RunTest<float>(std::string, ResultDatabase&, OptionParser&) (GEMM: GEMM.cpp,19)
2016/08/24 10:01:03.851	0.048193	0.169341	4,4,1	16,16,1	>>void RunTest<float>(std::string, ResultDatabase&, OptionParser&) (GEMM: GEMM.cpp,19)
2016/08/24 10:01:03.852	0.049633	0.174401	4,4,1	16,16,1	>>void RunTest<float>(std::string, ResultDatabase&, OptionParser&) (GEMM: GEMM.cpp,19)

....

```
openss>>expview -vfullstack
```

Exclusive Time (ms)	% of Total Exclusive Time	Exclusive Count	Call Stack Function (defining location)
11.818358	41.527561	240	main (GEMM: main.cpp,135) > @ 130 in RunBenchmark(ResultDatabase&, OptionParser&) (GEMM: GEMM.cpp,122) >> @ 240 in void RunTest<float>(std::string, ResultDatabase&, OptionParser&)
10.894840	38.282486	240	main (GEMM: main.cpp,135) > @ 137 in RunBenchmark(ResultDatabase&, OptionParser&) (GEMM: GEMM.cpp,122) >> @ 240 in void RunTest<double>(std::string, ResultDatabase&, OptionParser&)
2.992344	10.514553	60	main (GEMM: main.cpp,135) > @ 130 in RunBenchmark(ResultDatabase&, OptionParser&) (GEMM: GEMM.cpp,122) >> @ 231 in void RunTest<float>(std::string, ResultDatabase&, OptionParser&) (GEMM: GEMM.cpp,19)
2.753529	9.675400	60	main (GEMM: main.cpp,135) > @ 137 in RunBenchmark(ResultDatabase&, OptionParser&) (GEMM: GEMM.cpp,122) >> @ 231 in void RunTest<double>(std::string, ResultDatabase&, OptionParser&)

11 Memory Analysis Techniques

This O|SS version supports tracing memory allocation and deallocation function calls in user applications. This capability includes:

- Timeline of events that set a new high-water mark.
- List of event allocations (with calling context) to leaks.
- Overview of all unique callpaths to traced memory calls, providing max and min allocation and count of calls on this path.

The mem experiment supports sequential, MPI and threaded applications. No in-application instrumentation is needed. The mem experiment traces the following system calls:

- malloc
- calloc
- realloc
- free
- memalign
- posix_memalign

11.1 Memory Analysis Tracing (mem) experiment performance data gathering (ossmem)

To run the memory analysis experiment, use the ossmem convenience script and specify the application as an argument. No quotes are necessary if there are no arguments to the application, but they are placed here for consistency. In this example using the sweep3d application, the ossmem script will apply the memory analysis experiment by running the application with the O|SS memory trace collector, gathering the data and creating an O|SS database file containing experiment results. The performance information can be viewed via GUI or CLI.

```
# Sequential example:
ossmem "./lulesh2.0"
# MPI example:
ossmem "mpirun -np 64 ./sweep3d.mpi"
```

For example, here is a memory experiment run for the matmul application:

```
$ ossmem ./matmul
[openss]: mem using default experiment trace function list.
Creating topology file for frontend host localhost
Generated topology file: ./cbtfAutoTopology
Running mem collector.
Program: ./matmul
Number of mrnet backends: 1
Topology file used: ./cbtfAutoTopology
executing sequential program: cbtfrun -c mem --mrnet --openmp ./matmul
Main...
Do work...
Allocate matrix...
Allocate matrix...
Allocate matrix...
```



```

Initialize...
Initialize...
Initialize...
Compute...
Compute interchange...
Compute triangular...
Done.
All Threads are finished.
default view for /home/fred/sc16/exercises/matmul/matmul-mem-0.openss
[openss]: The restored experiment identifier is: -x 1
Performance data spans 19.928560 seconds from 2017/01/06 08:51:31 to 2017/01/06 08:51:50

```

Exclusive location) (ms)	% of Total Time	Number of Calls	Min Request Count	Min Requested Bytes	Max Request Count	Max Requested Bytes	Total Bytes Requested	Function (defining
0.013286	81.830500	1546	1	192	6	4096	6320832	_GI__libc_malloc (libc-2.17.so)
0.002144	13.205223	5						_GI__libc_free (libc-2.17.so)
0.000469	2.888643	7	1	368	1	368	2576	_calloc (libc-2.17.so)
0.000337	2.075634	1	1	72	1	72	72	_realloc (libc-2.17.so)

11.2 Viewing Memory Analysis Tracing (mem) experiment performance data via CLI

To launch the CLI on any experiment, use “openss -cli -f <database name>”. This table describes fields in the memory experiment default CLI view:

Column Name	Column Definition
Exclusive Mem Call Time	Aggregated total exclusive time spent in the memory function corresponding to this row of data.
% of Total Time	Percentage of exclusive time relative to the total time spent in the memory function corresponding to this row of data.
Number of Calls	Total number of calls to the memory function corresponding to this row of data.
Min Request Count	The number of times minimum bytes allocated or freed occurred during this experiment.
Min Requested Bytes	The minimum number of bytes that were allocated or freed by the corresponding memory function.
Max Request Count	The number of times maximum bytes allocated or freed occurred during this experiment.
Max Requested Bytes	The maximum number of bytes that were allocated or freed by the corresponding memory function.
Total Requested Bytes	The total number of bytes allocated by the corresponding function. Note: this does not subtract the bytes freed. This only totals the allocation function requested bytes.

Important command line interface (CLI) views are:

- **expview -vunique**

- Show times, call counts per path, min/max bytes allocation, total allocation to all unique paths to memory calls that the mem collector saw.
- **expview -vleaked**
 - Show function view of allocations that were not released while the mem collector was active.
- **expview -vtrace,leaked**
 - Will show a timeline of any allocation calls that were not released.
- **expview -vfullstack,leaked**
 - Display a full callpath to each unique leaked allocation.
- **expview -v trace,highwater**
 - Is a timeline of mem calls that set a new high-water mark.
 - The last entry is the allocation call that set the high-water mark for the complete run.
 - Investigate the last calls in the timeline and look at allocations that have the largest allocation size (size1,size2,etc) if your application is consuming lots of system RAM.

Here is a default view of the output from the ossmem experiment run of matmul on a small cluster. This shows the last eight allocation events that set the high-water mark:

```
openss>>expview -vtrace,highwater
Start Time(d:h:m:s) Event  Size Size Ptr  Return Value  New Call Stack Function (defining location)
                Ids  Arg1 Arg2 Arg  Highwater
*** trimmed all but the last 8 events of 61 ****
2016/11/10 09:56:50.824 11877:0 2080 0      0x7760e0 19758988 >>>>>>> _GI__libc_malloc (libc-2.18.so)
2016/11/10 09:56:50.826 11877:0 1728000 0      0x11783d0 21484908 >>>> _GI__libc_malloc (libc-2.18.so)
2016/11/10 09:56:50.827 11877:0 1728000 0      0x131e1e0 23212908 >>>> _GI__libc_malloc (libc-2.18.so)
2016/11/10 09:56:50.827 11877:0 1728000 0      0x14c3ff0 24940908 >>>> _GI__libc_malloc (libc-2.18.so)
2016/11/10 09:56:50.827 11877:0 2080 0      0x776a90 24942988 >>>>>>> _GI__libc_malloc (libc-2.18.so)
2016/11/10 09:56:50.919 11877:0 1728000 0      0x1654030 25286604 >>>> _GI__libc_malloc (libc-2.18.so)
2016/11/10 09:56:50.919 11877:0 1728000 0      0x17f9e40 27014604 >>>> _GI__libc_malloc (libc-2.18.so)
2016/11/10 09:56:50.919 11877:0 2080 0      0xabc6a0 27016684 >>>>>>> _GI__libc_malloc (libc-2.18.so)
```

Below is the default view of all unique memory calls seen while the **mem** collector was active. This is an overview of the memory activity. The default display is aggregated across all processes and threads. Users can view specific processes or threads.

For all memory calls the following are displayed:

- The exclusive time and percent of exclusive time.
- The number of times this memory function was called.
- The traced memory function name.

For allocation calls (e.g. malloc) the following are displayed:

- The maximum and minimum allocation size seen.

- The number of times that maximum or minimum was seen.
- The total size of all allocations.

```
openss>>expview -vunique
```

Exclusive (ms)	% of Total Time	Number of Calls	Min Request Count	Min Requested Bytes	Max Request Count	Max Requested Bytes	Total Bytes Requested	Function (defining location)
0.024847	89.028629	1546	1	192	6	4096	6316416	__GI__libc_malloc (libc-2.18.so)
0.002371	8.495467	5						__GI__libc_free (libc-2.18.so)
0.000369	1.322154	1	1	40	1	40	40	__realloc (libc-2.18.so)
0.000322	1.153750	3	1	368	1	368	1104	__calloc (libc-2.18.so)

NOTE: Number of Calls means the number of unique paths to the memory function call. To see the paths, use the CLI command: `expview -vunique,fullstack`

In this example, the sequential OpenMP version of lulesh was run under ossmem. The initial run detected 69 potential memory leaks. Examining the calltrees using the cli command "`expview -vfullstack,leaked -mtot_bytes`" revealed that allocations from the `Domain::Domain` constructor were not later released in the `Domain::~~Domain` destructor. Adding appropriate deletes in the destructor and rerunning ossmem found leaks detected in the `Domain` class were resolved. The remaining leaks were minor and from system libraries.

To see the improvements, use the `expstore` command to load the initial database and the second-run database, then use the `expcompare` cli command.

Below, database -x1 shows the initial run and -x2 shows results from the run with changes to address leaks detected in the `Domain` class:

```
openss>>expstore -f lulesh-mem-initial.openss
openss>>expstore -f lulesh-mem-improved.openss
openss>>expcompare -vleaked -mtot_bytes -mcalls -x1 -x2
```

-x 1, Total Bytes Requested	-x 1, Number of Calls	-x 2, Total Bytes Requested	-x 2, Number of Calls	Function (defining location)
10599396	69	3332	8	__GI__libc_malloc (libc-2.17.so)
72	1	72	1	__realloc (libc-2.17.so)

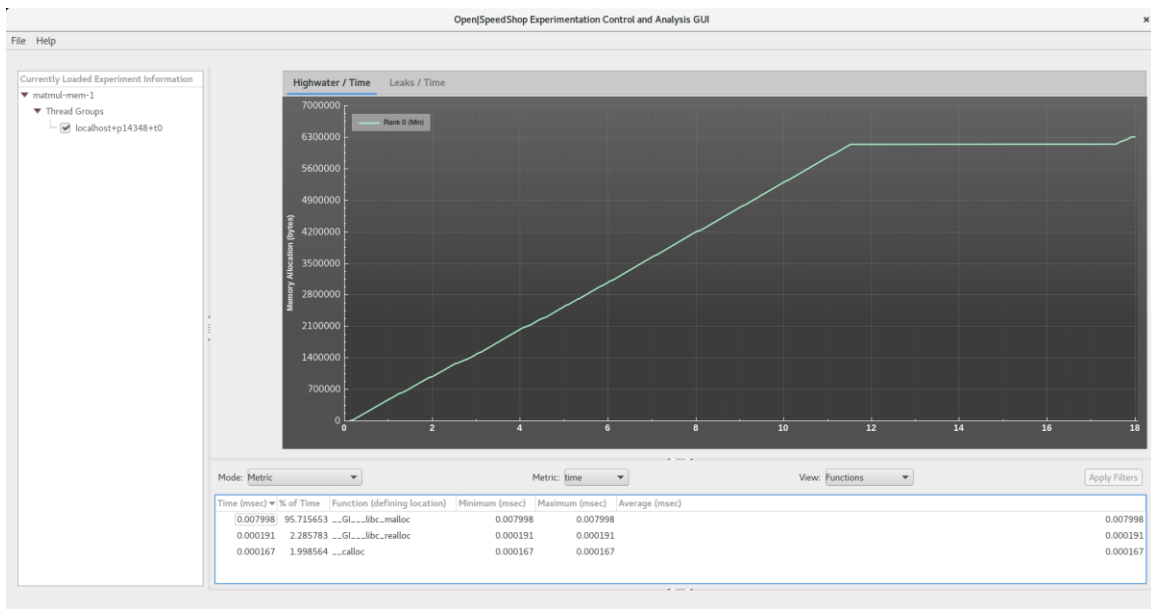
11.3 Viewing Memory Analysis Tracing (mem) experiment performance data via GUI

To launch the GUI on any experiment, use “openss -f <database name>”.

The first GUI display shown below is the default view for the mem experiment. It shows the memory functions that were called in the application, how many times they were called, the time spent in each and the percentage of the overall memory function time spent in each of the memory functions. This table identifies what each column represents in the default GUI view for the mem experiment:

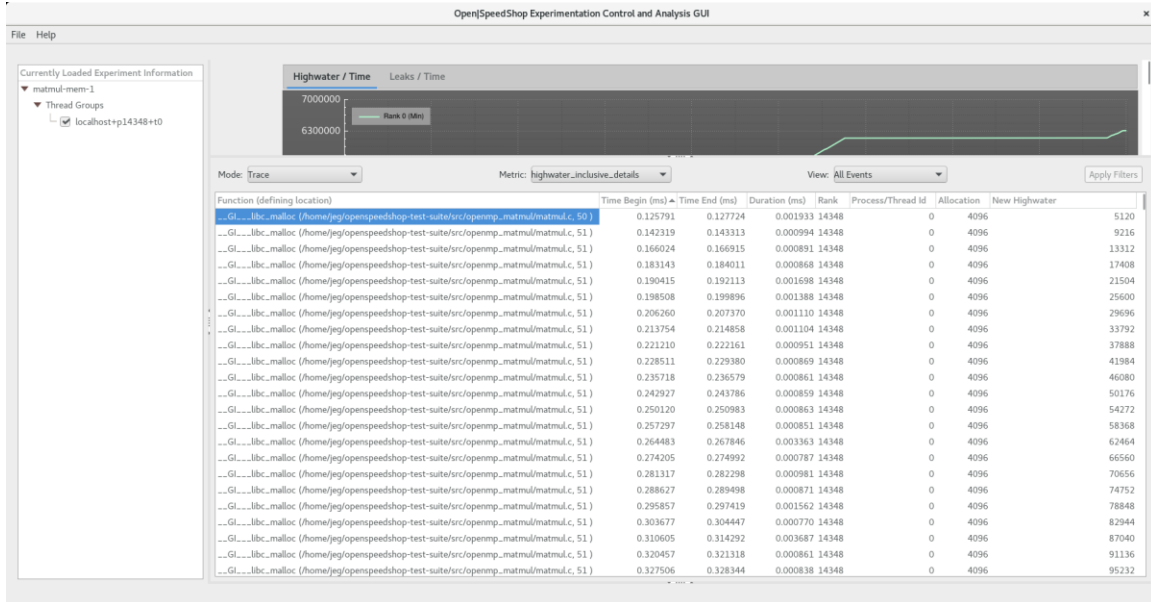
Column Name	Column Definition
Exclusive Mem Call Time	Aggregated total exclusive time spent in the memory function corresponding to this row of data.
% of Total Time	Percentage of exclusive time relative to the total time spent in the memory function corresponding to this row of data.
Number of Calls	Total number of calls to the memory function corresponding to this row of data.
Min Request Count	The number of times minimum bytes allocated or freed occurred during this experiment.
Min Requested Bytes	The minimum number of bytes that were allocated or freed by the corresponding memory function.
Max Request Count	The number of times maximum bytes allocated or freed occurred during this experiment.
Max Requested Bytes	The maximum number of bytes that were allocated or freed by the corresponding memory function.
Total Requested Bytes	The total number of bytes allocated by the corresponding function. Note: this does not subtract the bytes freed. This only totals the allocation function-requested bytes.

The paths to each memory function call, through the source, are available through the call path views. This is the high-water memory experiment GUI view for the matmul application:

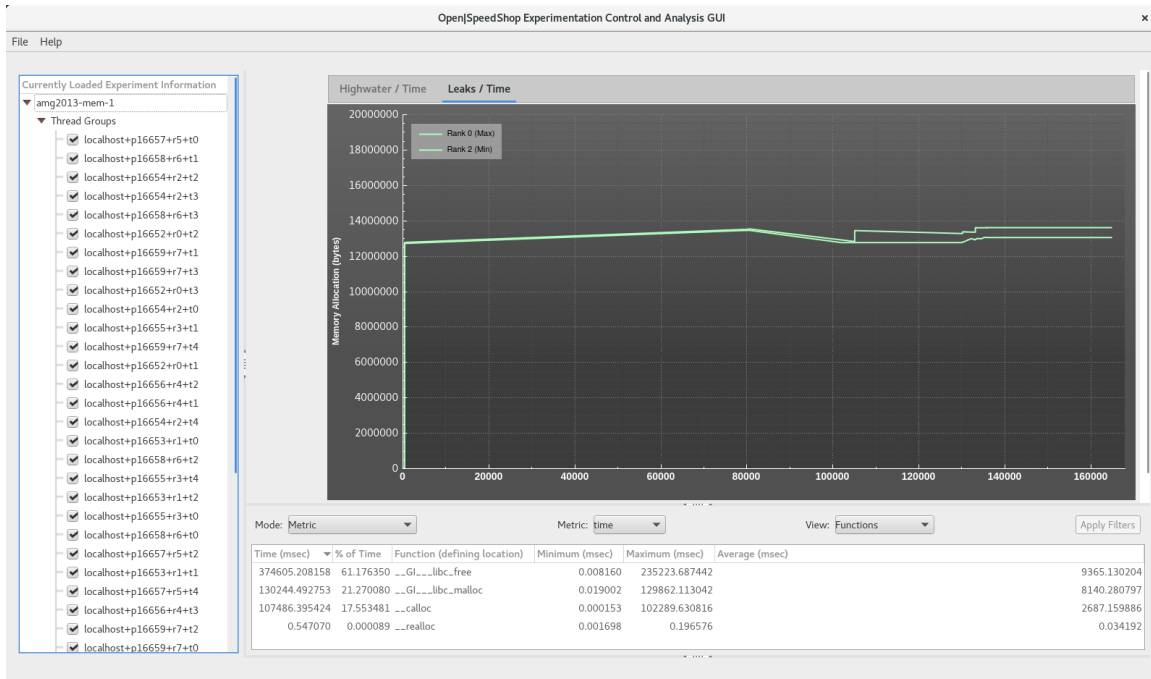


There are other GUI views based on the high-water mark, Unique call paths and Leaked memory. The CLI will show the same information.

Here is the trace of the memory calls that changed the high-water value as the application executed.



The view below shows the memory leaks that occurred while running amg2013. The minimum and maximum leaks are tracked in the graphical view.



Views mentioned above are accessed through the openss-gui -f <database name> command.

12 Advanced Analysis Techniques

Analyzing the results of a single performance experiment can be useful for debugging and tuning a code, but comparing results of different experiments can show users how an application's performance has changed. This is useful for tracking how performance varies with each version of an application or for understanding how a different compiler or compiler options affect an application's performance. This also lets users perform scalability tests to see how their application's performance scales with the number of processors. It's also helpful just to see the progress made while tuning a code.

O|SS has options that let users compare performance data. Use the Custom Compare Panel (CC icon) in the GUI or the osscompare convenience script:

```
> osscompare "db1.openss, db2.openss, ..." [options]
```

This will produce side-by-side comparison listings of up to eight databases at once. See the osscompare man page for more details. Here is an example comparing two pcsamp experiments on the smg2000 application:

```
osscompare "smg2000-pcsamp.openss, smg2000-pcsamp-1.openss"
```

```
[openss]: Legend: -c 2 represents smg2000-pcsamp.openss
```

[openss]: Legend: -c 4 represents smg2000---pcsamp---1.openss		
-c 2, Exclusive CPU	-c 4, Exclusive CPU	Function (defining location)
time in seconds.	time in seconds.	
3.870000000	3.630000000	hypr_SMGResidual (smg2000:smg_residual.c,152)
2.610000000	2.860000000	hypr_CyclicReduction (smg2000:cyclic_reduc;on.c,757)
2.030000000	0.150000000	opal_progress (libopen-pal.so.0.0.0)
1.330000000	0.100000000	mca_btl_sm_component_progress (libmpi.so.0.0.2)
0.280000000	0.210000000	hypr_SemiInterp (smg2000: semi_interp.c,126)
0.280000000	0.040000000	mca_pml_ob1_progress (libmpi.so.0.0.2)

12.1 Comparison Script Argument Description

The O|SS comparison script accepts a number of arguments. This section describes acceptable options for those arguments. For a quick overview, see *14.4 osscompare: Compare Database Files*. As described above, the osscompare script accepts at least two and up to eight comma-separated database file names, enclosed in quotes as the mandatory argument. By default, the compared metric is the primary one the experiment produced. For most experiments, this is exclusive time, but hardware counter experiments use the number of hardware counter overflows. These are the default or mandatory arguments to osscompare. The following sections describe arguments for osscompare in more detail.

12.1.1 osscompare metric argument

The osscompare metric argument specifies the type of performance information O|SS will use to compare against when examining each database file in the compare database file list. To find the legal metric specifications and produce comparison outputs, open one of the database files with the O|SS command line interface (CLI) and list the available metrics:

```
openss -cli -f smg2000-pcsamp.openss
openss>>list -v metrics
pcsamp::percent
pcsamp::threadAverage
pcsamp::threadMax
pcsamp::threadMin
pcsamp::time
```

Use the output of the list metrics command as an argument to the osscompare command as shown in these examples:

```
osscompare "smg2000-pcsamp.openss,smg2000-pcsamp-1.openss"
osscompare "smg2000-pcsamp.openss,smg2000-pcsamp-1.openss" percent
osscompare "smg2000-pcsamp.openss,smg2000-pcsamp-1.openss" threadMin
osscompare "smg2000-pcsamp.openss,smg2000-pcsamp-1.openss" threadMax
```

There are exceptions. For example, some experiments such as usertime and hwtime have “details”-type metrics output by the list metrics CLI command (list -v metrics). These will not work as a metric argument to osscompare.

For the hwc and hwctime hardware counter experiments, use the actual PAPI event name in addition to the metric names output from the list metric command. This example database file was generated using the PAPI_TOT_CYC event:

```
openss -cli -f smg2000-hwc.openss
openss>>[openss]: The restored experiment identifier is: -x 1
openss>>list -v metrics
hwc::overflows
hwc::percent
hwc::threadAverage
hwc::threadMax
hwc::threadMin
```

Here are a couple of osscompare examples where “hwc::overflows” can be used interchangeably with PAPI_TOT_CYC:

```
osscompare "smg2000-hwc.openss,smg2000-hwc-1.openss" hwc::overflows
osscompare "smg2000-hwc.openss,smg2000-hwc-1.openss" PAPI_TOT_CYC
```

Note: For compares involving hwcsamp metric-based databases, use the “allEvents” metric in the osscompare command to compare all of the existing hardware counters from each experiment. That will compare all events in each of the databases and will ignore the program counter sampling data from each of the databases. The form of the osscompare command to compare all the hardware counter events is:

```
osscompare "smg2000-hwcsamp.openss,smg2000-hwcsamp-1.openss" allEvents
```

12.1.2 osscompare rows of output argument

The osscompare command allows the user to specify how many lines of the comparison output to generate. The argument is optional

"rows=nn" is defined as follows:
"nn" - Number of rows/lines of performance data output.

In this next example, only ten (10) lines of comparison will be shown when the osscompare command is executed. It will be the most interesting, or top, ten lines:

```
osscompare "smg2000-hwc.openss,smg2000-hwc-1.openss" hwc::overflows rows=10
```

12.1.3 osscompare output name argument.

The osscompare command allows the user to specify the name to be used when writing out the comparison output files. The argument is optional.

"oname=<output file name>" is defined as follows:
"output file name" - Name given to the output files created for the comparison.

This argument is valid when the environment variable OPENSS_CREATE_CSV is set to 1. In this example, the comparison files created when the osscompare command is executed will be named smg_hwc_cmp.csv and/or smg_hwc_cmp.txt:

```
osscompare "smg2000-pcsamp.openss,smg2000-pcsamp-1.openss" oname=mar2015_pcsamp_cmp
```

This example will generate comparison files named using the particular oname specification:

```
8 -rw-rw-r--. 1 fred fred 4475 Mar 11 15:53 mar2015_pcsamp_cmp.compare.csv
8 -rw-rw-r--. 1 fred fred 4841 Mar 11 15:53 mar2015_pcsamp_cmp.compare.txt
```

12.1.4 osscompare view type or granularity argument.

The osscompare command allows an optional view type argument representing the granularity. O|SS allows for viewing performance data at three levels: linked object, function and statement. The osscompare command will produce output at one of those levels based on the view type argument where:

```
"viewtype=<functions | statements | linkedobjects >" is defined as follows:
    "functions"      - View type granularity is per function
    "statements"     - View type granularity is per statement
    "linkedobjects"  - View type granularity is per library (linked object)
```

The following example will produce a side-by-side comparison for the statement level, not the default function level. So, this example will compare statement performance values in each of the two databases and produce a side-by-side comparison showing how each statement in the application differed from the two experiments:

```
osscompare "smg2000-pcsamp.openss,smg2000-pcsamp-1.openss" viewtype=statements
```

13 O|SS User Interfaces

The O|SS (O|SS) GUI has been used throughout this manual and users are encouraged play with the interface to become familiar with it. The GUI lets users peel off and rearrange any panel. There also are context-sensitive menus, letting users right-click on any location to access a different view or activate additional panels.

For users who prefer not to employ the GUI, there are three other options with equal functionality. First there is the command line interface that has been illustrated throughout this manual. It is launched with the -cli option:

```
> openss -cli
```

There also is the immediate command (batch) interface. This uses the -batch flag:

```
> openss -batch < openss_cmd_file
> openss -batch -f <exe> <experiment>
```

Lastly, there is a Python scripting API, letting users launch O|SS commands within a python script:

```
> python openss_python_script_file.py
```

13.1 Command Line Interface Basics

The interactive command line interface offers processing like gdb or dbx. Several interactive commands allow users to create experiments, provide users with process/thread control or enable users to view experiment results. Full CLI documentation is available at http://www.openspeedshop.org/doc/cli_doc/, but some important points are briefly covered here. This is a quick overview of some commands (those marked with * are only available for the online version):

Experiment Creation <ul style="list-style-type: none"> • expcreate • expattach* 	Result Presentation <ul style="list-style-type: none"> • expview • opengui
Experiment Control <ul style="list-style-type: none"> • expgo • expwait* • expdisable* • expenable* 	Misc. Commands <ul style="list-style-type: none"> • help • list • log • record • playback • history • quit
Experiment Storage <ul style="list-style-type: none"> • expsave • expstore 	

This is a simple example to create, run and view data from an experiment using the CLI:

> openss -cli	Open the CLI.
openss>> expcreate -f "mutatee 2000" pcsamp	Create an experiment using pcsamp with this application.
openss>> expgo	Run the experiment and create the database
openss>> expview	Display the default view of the performance data.

Users also can get alternative views of the performance data within the CLI. Here is a list of some options to change the way the information is displayed:

help or help commands	Display CLI help text. ^{1 1 1 1} _{SEP}
expview	Show the default view for experiment.
expview -v statements	Show time-consuming statements. ^{1 1 1 1} _{SEP}

expview -v loops	Show time-consuming loop.
expview -v vectorinstr	On Intel platforms: show instructions that are vector and the time spent in vector instruction execution
expview -v linkedobjects	Show time spent in libraries.
expview -v fullstack	See all unique call paths in the application.
expview -m loadbalance	See load balance across all the ranks/threads/processes in the experiment.
expview -r <rank_num>	See data for specific rank(s)
expcompare -r 1 -r 2 -m time	Compare rank 1 to rank 2 for metric equal to "time". Other metrics are allowed. This is a usage example.
list -v metrics	See the list of optional performance data metrics.
list -v src	See the list of source files associated with experiment.
list -v obj	See the list of object files associated with experiment.
list -v ranks	See the list of ranks associated with experiment.
list -v hosts	See machine host names associated with experiment.
expview -m <metric>	See performance data for the specified metric .
expview -v fullstack <experiment type> <number>	See <number> of call paths from the list of expensive call paths.
expview -v fullstack usertime2	Shows the top two call paths in execution time.
expview <experiment-name><number>	Shows <number> of the functions from the list of the top time-consuming functions.
expview pcsamp2	Shows the two functions consuming the most time.
expview -v statements <experiment-name><number>	Show <number> of the statements from the list of the top time-consuming statements.
expview -Fcsv	Show the view in comma separated list format (csv)

Remember, to use the GUI at any time just issue the command **opengui** in the CLI.

13.1.2 CLI Metric Expressions and Derived Types

O|SS can create a derived metric from the gathered metrics by using the metric expression math functionality in the command line interface (CLI). Access the overview from the CLI by typing this help CLI command:

```
openss>>help metric_expression
```

```
*****
```

```
<metric_expression> ::= <string> ( [ <constant> || <metric_expression> ] [,
[ <constant> || <metric_expression> ] ]*)
```

A user-defined expression that uses metrics to compute a special value for display in a report.

User-defined expression can be added to an <expMetric_list>.

A functional notation is used to build the desired expression and the following simple arithmetic operations are available:

Function	# arguments	returns
Uminus()	1	unary minus of the argument
Abs()	1	Absolute value of the argument
Add()	2	summation of the arguments
Sub()	2	difference of the arguments
Mult()	2	product of the arguments
Div()	2	first argument divided by second

Mod()	2	remainder of divide operation
Min()	2	minimum of the arguments
Max()	2	maximum of the arguments
A_Add()	1	sum of all the data samples specified for the view
A_Mult()	1	product of all the data samples specified for the view
A_Min()	1	minimum of all the data samples specified for the view
A_Max()	1	maximum of all the data samples specified for the view
Sqrt()	1	square root of the argument
Stdev()	3	standard deviation calculation
Percent()	2	percent the first argument is of the second
Condexp()	3	"C" expression: "(first argument) ? second argument: third argument"
Header()	2	use the first argument as a column header for the display of the second

Note:

Integer and floating constants are supported as arguments as are the metric keywords associated with the experiment view.

Arguments to these functions can be <metric_expressions>, with the exception of the first argument of 'Header'.

The first argument of 'Header' must be a character string that is preceded with and followed by '\ '.

When the '-v summary' option is used, it is not generally possible to produce a meaningful column summary. A summary is produced for Add(), Max(), Min(), Percent(), A_Add(), A_Max and A_Min().

Examples:

```
expview hwc -m count,Header("\percent of counts\","Percent(count,A_Add(count)) -v summary
expview mpi -v butterfly -f MPI_Alltoallv -m time,Header("average time/count",Div(Mult(time,1000),counts))
expview -m papi_l2_tca,papi_l2_tcm,Header("\percent of l2_tcm/l2_tca\","Percent(papi_l2_tcm,papi_l2_tca))
```

The following example for study takes the default view, expview command, and appends the capability to add the percentage each function contributes to the total.

Use the "Header" phrase to create a header for the new data column being added. Use the "Percent" phrase to create the arithmetic expression that divides the PAPI_L1_DCM counts (count) for each function by the total number of PAPI_L1_DCM counts in the application(A_Add(count)):

```
openss>>expview -m count,Header("\percent of counts\","Percent(count,A_Add(count)))
```

Exclusive	percent	Function (defining location)
PAPI_L1_DCM	of counts	
Counts		
342000000	52.333588	hypr_SMGResidual (smg2000: smg_residual.c,152)
207500000	31.752104	hypr_CyclicReduction (smg2000: cyclic_reduction.c,757)
20500000	3.136955	hypr_SemiInterp (smg2000: semi_interp.c,126)
15000000	2.295333	hypr_SemiRestrict (smg2000: semi_restrict.c,125)
8500000	1.300689	pack_predefined_data (libmpi.so.0.0.3)
7000000	1.071155	unpack_predefined_data (libmpi.so.0.0.3)

Another example, this one based in the hwcsamp experiment view, shows the ratio between total cache accesses and total cache misses. A header is created, defined by the Header clause:

```
openss>>expview -m papi_l2_tca,papi_l2_tcm,Header("\percent of l2_tcm/l2_tca\","Percent(papi_l2_tcm,papi_l2_tca))
```

papi_l2_tca	papi_l2_tcm	percent of l2_tcm/l2_tca	Function (defining location)
289946516	109226440	37.671237	hypr_SMGResidual (smg2000: smg_residual.c,152)
203463495	74795126	36.760956	hypr_CyclicReduction (smg2000: cyclic_reduction.c,757)
34442810	12746112	37.006597	mca_btl_vader_check_fboxes (libmpi.so.1.4.0: btl_vader_fbox.h,108)
25522126	8311723	32.566734	hypr_SemiInterp (smg2000: semi_interp.c,126)
...			
...			

13.1.3 CLI Automatically Generated Derived Metrics and CLI Derived Metric Names

The CLI view code has logic to match up existing PAPI hardware counters with other PAPI hardware counters, if the user specified a combination of counters that OJSS has been coded to recognize. Users have requested these combinations of counters as interesting ones to have pre-computed and output in the CLI views.

The list of automatically generated and displayed derived metric values are discussed in the following paragraphs. Here is a short list of the metric names that can be used in the CLI view and the hardware counters needed:

Purpose/Function:	CLI metric name:	PAPI formula:
Computational Intensity	-m intensity	PAPI_TOT_INS/PAPI_TOT_CYC
Reports level 1 data cache to total cache miss ratio	-m l1dcmiss	PAPI_L1_DCM/PAPI_L1_TCA
Reports L1 data cache read miss ratios	-m l1dcrmiss	PAPI_L1_DCM / PAPI_L1_DCA
Reports level 2 data cache miss ratio	-m l2dcmiss	PAPI_L2_DCM / PAPI_L2_TCA
Reports level 2 cache miss ratio	-m l2tcmiss	PAPI_L2_TCM/PAPI_L2_TCA
Reports L2 cache data hit rate	-m l2dchitrate	(1.0 - (PAPI_L2_DCM / PAPI_L1_DCA))
Reports level 3 total cache to cache access ratio	-m l3tcmiss	PAPI_L3_TCM/PAPI_L3_TCA
Reports level 3 total cache to data cache access ratio	-m l3tdcmiss	PAPI_L3_TCM / PAPI_L3_DCA
Reports data references per instruction	-m datarefperinstr	PAPI_L1_DCA / PAPI_TOT_INS
Reports double precision flops	-m dflops	PAPI_DP_OPS / time
Reports single precision flops	-m flops	PAPI_FP_OPS / time

Reports ratio of floating point instructions to total instructions	-m fpinstratio	PAPI_FP_INS / PAPI_TOT_INS
Reports graduated floating point instructions per cycle	-m gradfpinst	PAPI_FP_INS / PAPI_TOT_CYC
Reports ratio of mis-predicted to correctly predicted branches	-m mispredicted	PAPI_BR_MSP / PAPI_BR_PRC
Reports SIMD_FP_256:packed_single / PAPI_FP_OPS	-m simdfpfpops	SIMD_FP_256:packed_single/PAPI_FP_OPS
Reports SIMD_FP_256:packed_double / PAPI_DP_OPS	-m simdfpdpops	SIMD_FP_256:packed_double/PAPI_DP_OPS

13.1.3.1 Computational Intensity

For this derived metric, a ratio is created based on the number of total instructions (PAPI_TOT_INS) divided by the PAPI_TOT_CYC hardware counter value. This ratio gives an idea of the instruction execution computational intensity. TBD.

13.1.3.2 Level 1 Data Cache Miss Ratio

For this derived metric, a ratio is created based on the number of total level 1 cache accesses with the level 1 data cache misses. This ratio gives... TBD.

13.1.3.3 Level 2 Data Cache Miss Ratio

For this derived metric, a ratio is created based on the number of total level 2 cache accesses with the level 2 data cache misses. This ratio gives... TBD.

13.1.3.4 Level 3 Data Cache Miss Ratio

For this derived metric, a ratio is created based on the number of total level 3 cache accesses with the level 3 data cache misses. This ratio gives... TBD.

13.2 CLI Batch Scripting

Users with a known set of commands they want to issue can create a plain text file with CLI commands. For example, here's a batch file that will create, run and view the pcsamp experiment on the application fred:

```
# Create batch file commands
> echo expcreate -f fred pcsamp >> input.script
> echo expgo >> input.script
> echo expview pcsamp10 >> input.script
```

To run the batch file input.script, use the `-batch` option to openss:

```
> openss -batch < input.script
```

Note that in this context this interface is only supported via the online version of O|SS, so it must have been built with the `OPENSS_INSTRUMENTOR=mrnet` options.

13.3 Python Scripting

The O|SS Python API lets users execute the same interactive/batch commands directly through Python. Users can intersperse the normal Python code with commands to O|SS. Currently this interface is only supported via the online O|SS version.

13.4 MPI_Pcontrol Support

O|SS also supports the `MPI_Pcontrol` function. This feature lets the user gather performance data only for sections of their code bounded by `MPI_Pcontrol` calls.

The `MPI_Pcontrol` must be added to the application's source code.

`MPI_Pcontrol(1)` enables performance data gathering; `MPI_Pcontrol(0)` disables it.

Users also must set the O|SS environment variable `OPENSS_ENABLE_MPI_PCONTROL` to 1 to activate the `MPI_Pcontrol` call recognition. Otherwise the `MPI_Pcontrol` statements will be ignored.

Users can optionally set the `OPENSS_START_ENABLED` environment variable to 1 to gather performance data until an `MPI_Pcontrol(0)` call is encountered.

If `OPENSS_START_ENABLED` is not set, **no performance data will be gathered until an `MPI_Pcontrol(1)` call is encountered.**

Note that for `OPENSS_START_ENABLED` to have any effect, `OPENSS_ENABLE_MPI_PCONTROL` must be set.

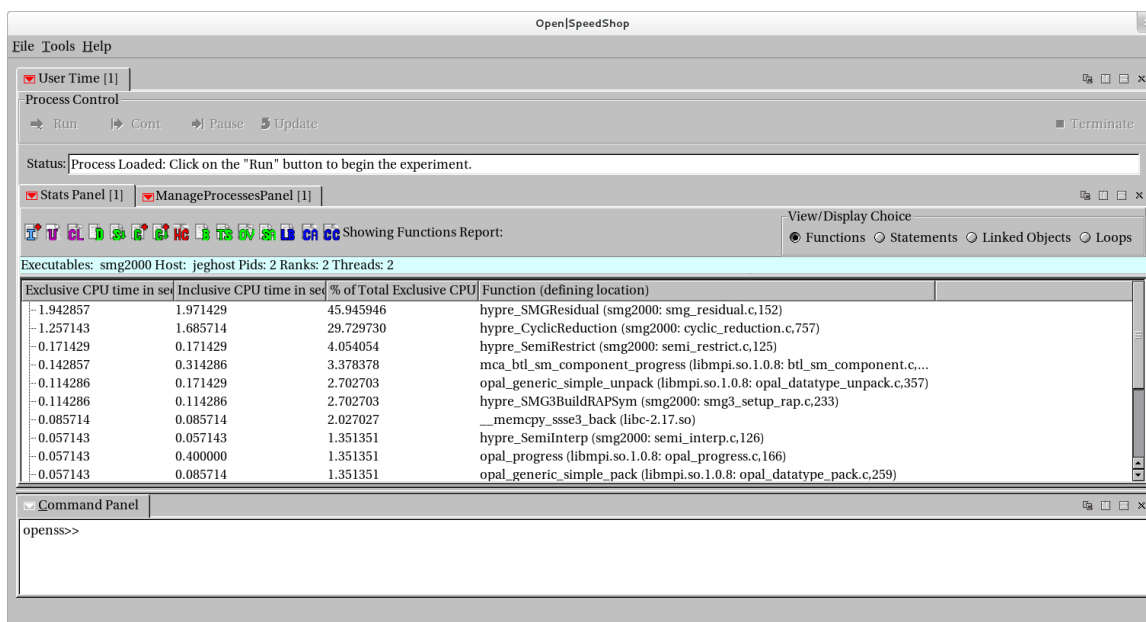
13.5 Qt3 Legacy Graphical User Interface Basics

This section gives an overview of the O|SS graphical user interface with a focus on the GUI's basic functionality.

To launch the GUI on any experiment, use "`openss -f <database name>`".

13.5.1 Basic Initial View – Default View

The usertime experiment default view is used here as an illustration because it has many of the icons and features of other OJSS experiments:



13.5.1.1 Icon ToolBar



The most used items, found in the Stats Panel menu located under the Stats Panel tab, also are available in the Stats Panel Toolbar, which is provided as a convenience. A quick overview of the toolbar options is below. Toolbar contents vary by experiment, because some options don't make sense for all experiments. This table describes the icons and the functions they represent:

"I"	Information	This shows the experiment metadata. Information such as the experiment type, processes, ranks, threads, hosts and other experiment-specific information is displayed.
"U"	Update	This updates information in the Stats Panel display. This can be used to display any new data that may have come from the nodes on which the application is running.
"CL"	Clear auxiliary information	If the user has chosen a performance data time segment or a specific function for which to view the data, this option clears those settings and allows the next view selection to show data for the entire program again.
"D"	Default View	The default view icon shows performance results based on the view choice granularity selection.

"S, down arrow"	Statements per Function	Show performance results related back to the source statements in the application for the selected function. Highlight a function in the Stats Panel and click on this icon.
"C-plus sign"	Call paths w/o coalescing	Show all the calling paths in this application. Duplicate paths will not be coalesced. All the calling paths will be shown in their entirety.
"C-plus sign, down arrow"	Call paths w/o coalescing per Function	Show all calling paths in this application for only the selected function. Highlight a function in the Stats Panel and click on this icon. Duplicate paths will not be coalesced. All the calling paths will be shown in their entirety.
"HC"	Hot Call Path	Show the call path in the application that took the most time. This is a short cut to find the "hot" call path.
"B"	Butterfly view	Show the butterfly view, which displays the callers and callees of the selected function. Highlight a function in the Stats Panel and click on this icon, then repeat to drill down into the callers and/or callees.
"TS"	Time Segment	Show a portion of the performance data results based on the time segment selected.
"OV"	Optional View	Use this dialog to select which performance metrics to show in the new performance data report.
"SA"	Source Annotation	Choose which metric to use in the source panel to annotate the source. Defaults are different for each experiment, but usually is "time".
"LB"	Load Balance	Show the load balance view, which displays the min, max and average performance values for the application; only available on threaded or multiple-process applications.
"CA"	Cluster Analysis	Show the comparative analysis view, which displays the output of a cluster analysis algorithm run against the threaded or multiple-process performance analysis results for the user application. This view's use is to find outlying threads or processes and report the groups of like-performing threads, processes or ranks.
"CC"	Custom Compare	Raise the custom comparison panel, which provides mechanisms allowing the user to create custom views of the performance analysis results. This lets the user supplement the provided O SS views.

13.5.1.2 View/Display Choice Selection

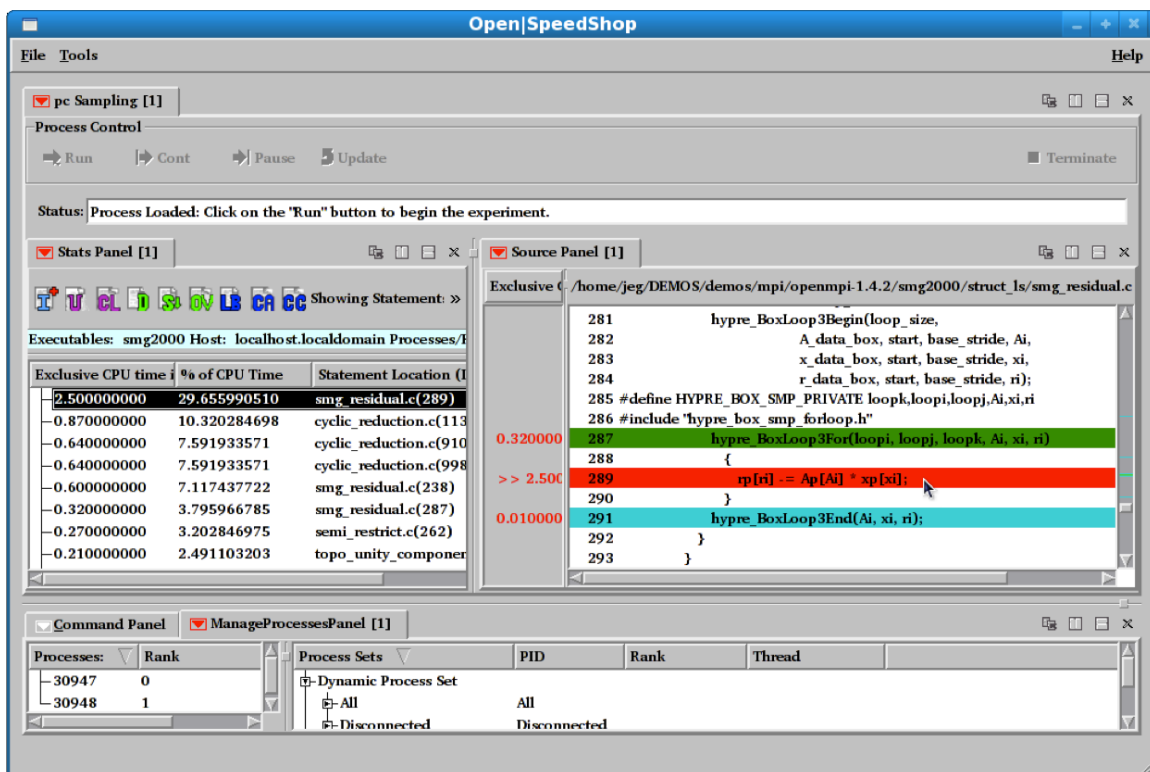
The View/Display Choice set of buttons lets users choose the granularity for a particular display. Normally a user chooses a view choice granularity and then selects a view via one of the icons described in the table above. The choices, as shown in the image below, are:

- Per Function – Display the performance information relative to each function in the program that had performance data gathered during the experiment.
- Per Statement – Display the performance information relative to each statement in the program that had performance data gathered during the experiment.
- Per Linked Object – Display the performance information relative to each library or linked object in the program that had performance data gathered during the experiment.

- Per Loop – Display the performance information relative to each loop in the program that had performance data gathered during the experiment. Note that the loop performance information is shown only for loops that actually were executed. There may be loops in the application that will not show up in the display because they were not executed or had minimal time attributed to them.



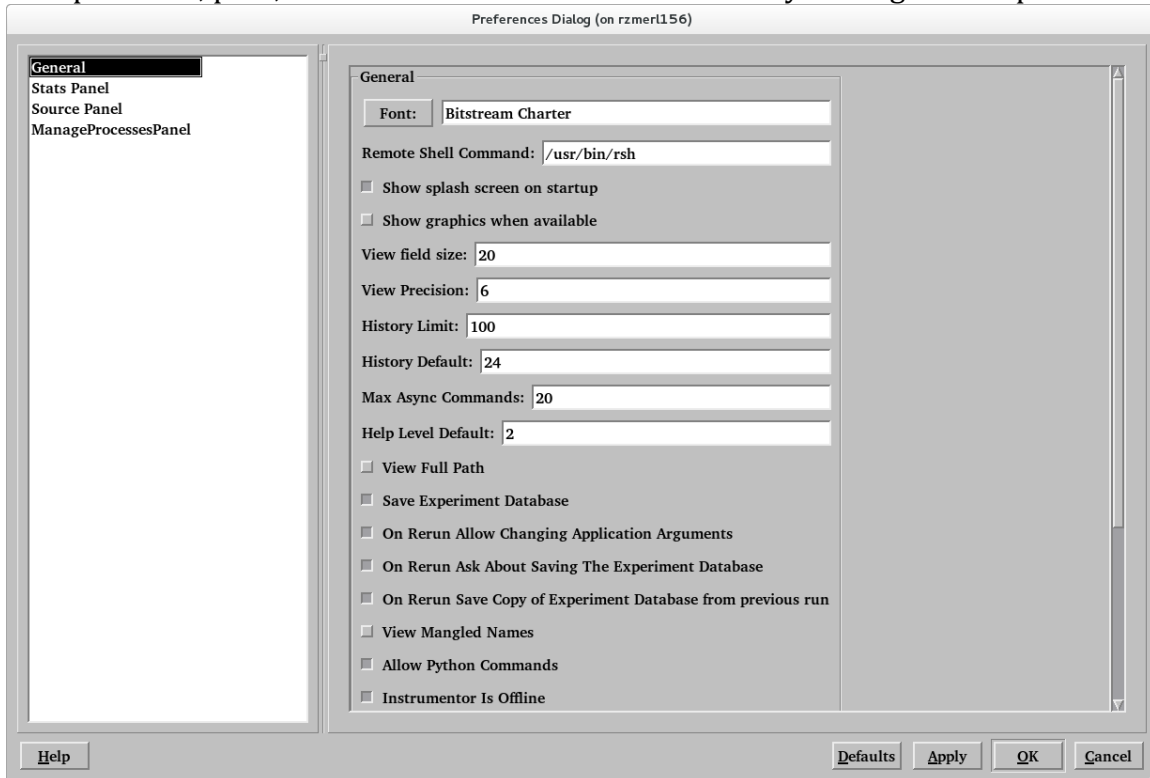
The image below shows that double-clicking on a line of statistical information in the Stats Panel will focus the source panel at the line of source representing the performance information and annotates the source with that information. Note the hot (red) to cold (blue) color highlights: The higher the performance values, the hotter the color. Source highlighted in red takes the most time in the profiled program:



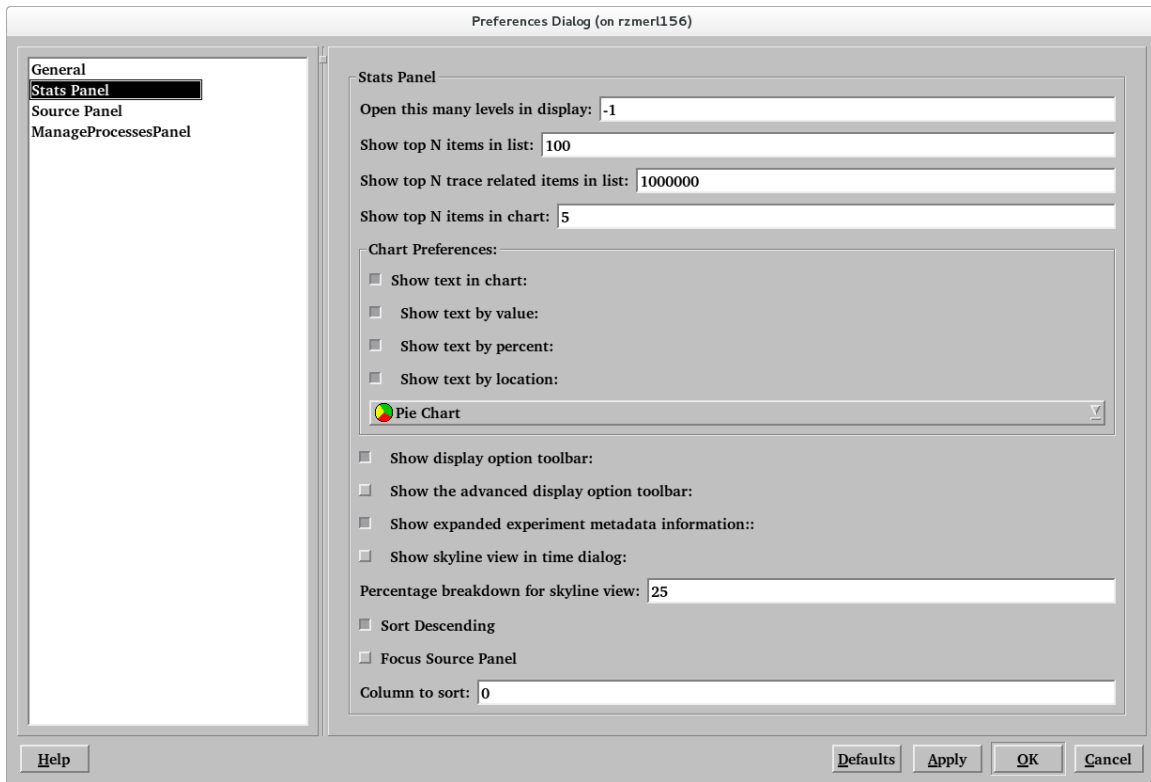
13.5.2 Preferences - How to change preferences

These preference panel images are included to outline the sequence for changing the GUI and CLI options to generate and view performance information. The first

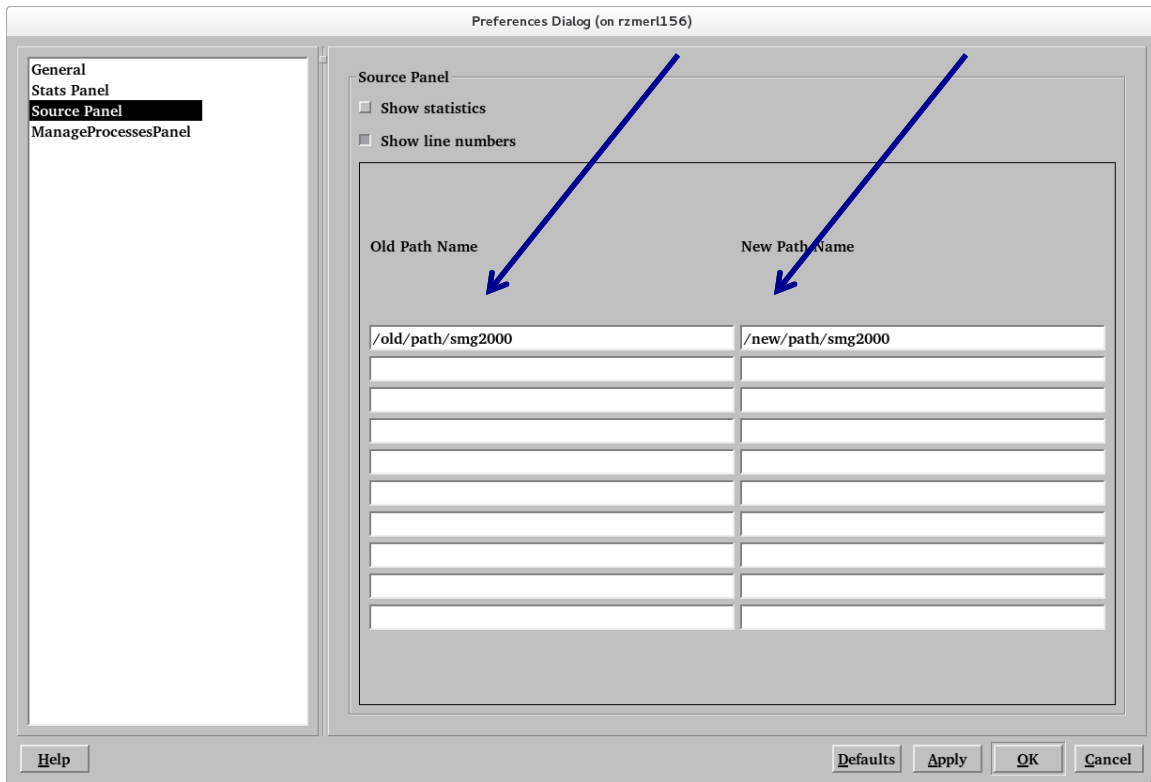
view is the main (General) preference panel, which sets the font, view field sizes, data precision, path, number of lines in the view and many other general options:



The Stats Panel preference panel lets users change preferences for viewing the performance information in the GUI Stats Panel:

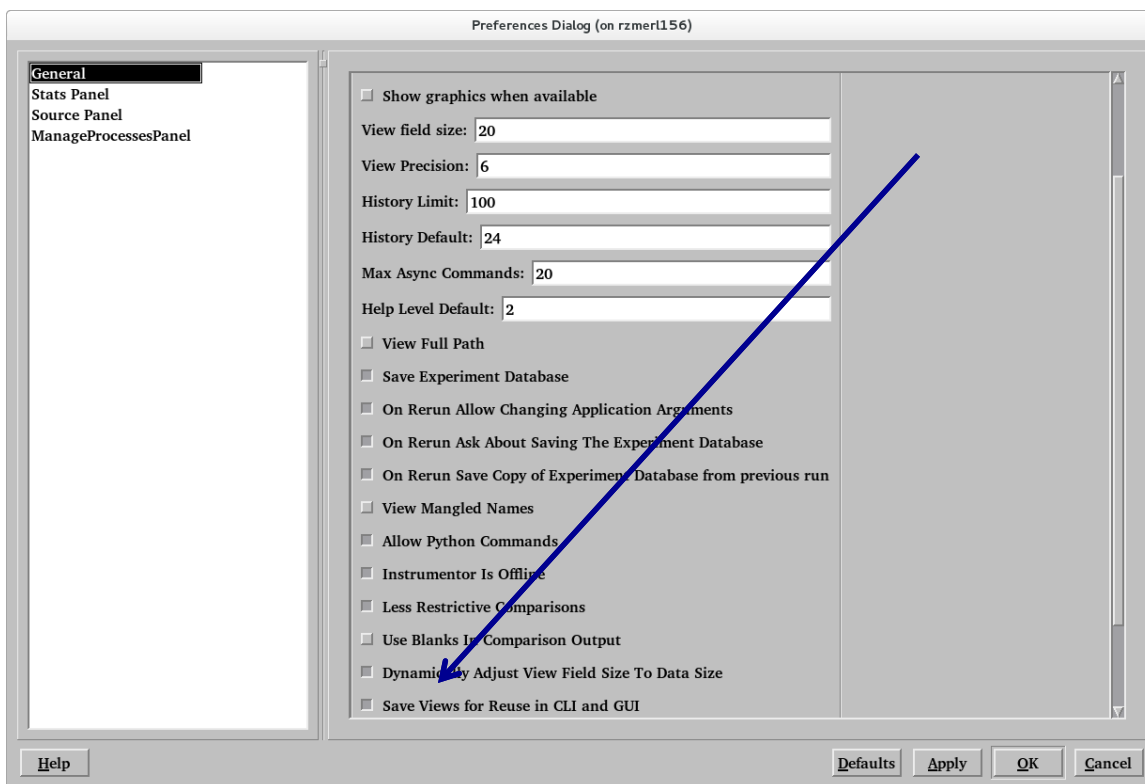


The Source Panel preference panel lets users remap paths to source files that are in a different location on the viewing platform. Use this when the source files on the viewing machine aren't visible because the executable was built on a different machine. Put the old path to the source into the Old Path Name text box area and put the new path for the source on the viewing machine into the New Path Name text box area.



13.5.2.1 Disabling or enabling the preference for Save/Reuse views in CLI.

This shows the General preferences window, scrolled down to the area that shows more preference options. Users who do not want an active new save/reuse view can click on the “Save Views for Reuse in CLI and GUI” (see the blue arrow below) to disable that function. Clicking on that preference line disables or enables the feature. This same procedure also works for the other preferences.



13.6 Next Generation O|SS GUI Application

13.6.1 Introduction

Originally developed as the Graphical User Interface (GUI) for NVIDIA CUDA application performance analysis under a NASA SBIR contract, the Next Generation O|SS GUI has been expanded to include support for other O|SS sampling and tracing experiments such as “pcsamp”, “usertime”, “hwc”, “hwctime”, “hwcsamp”, “omptp”, “mem”, “io”, “iop”, “iot”, “mpi”, “mpip”, “mpit” and “pthreads”.

The Next Generation O|SS GUI application, having the executable name “openss-gui”, allows the user to explore application experiment trace data within in a timeline graph view or hardware performance counter data within line or bar graph views with additional details shown in a table view and correlated to the source-code as applicable.

In general, to launch the Next Generation O|SS GUI application for any experiment, use:

```
“openss-gui [-f <database name>]“
```

The “-f <database name>” command-line option is optional as indicated by the brackets “[...]”. NOTE: The brackets are not entered by the user it only indicates optional entry.

From the command-line general application usage instructions may be viewed by using the “--help” or “-h” command-line option:

```
$ ./openss-gui --help
Usage: ./openss-gui [options]
Open|SpeedShop Application Performance Analysis GUI

Options:
  -h, --help           Displays this help.
  -v, --version         Displays version information.
  -f, --file <file>    The Open|SpeedShop experiment database (.openss) file to
                        load.
```

Just to reiterate this is a different GUI than the original O|SS GUI based on Qt3. The Next Generation O|SS GUI is launched by using “openss-gui” instead of “openss”. The Qt3 GUI can still be activated by running “openss” without the “-cli” command-line option.

13.6.1.1 Main Window User Interface Layout

The application user interface is laid out in a logical manner to present a comprehensive view of the performance characteristics of an application. The main screen of the application is divided into four sections (ref. *Figure 1, “Main Window User Interface Layout”*).

- Experiment Panel
- Metric Plot View
- Metric Table View
- Source Code View

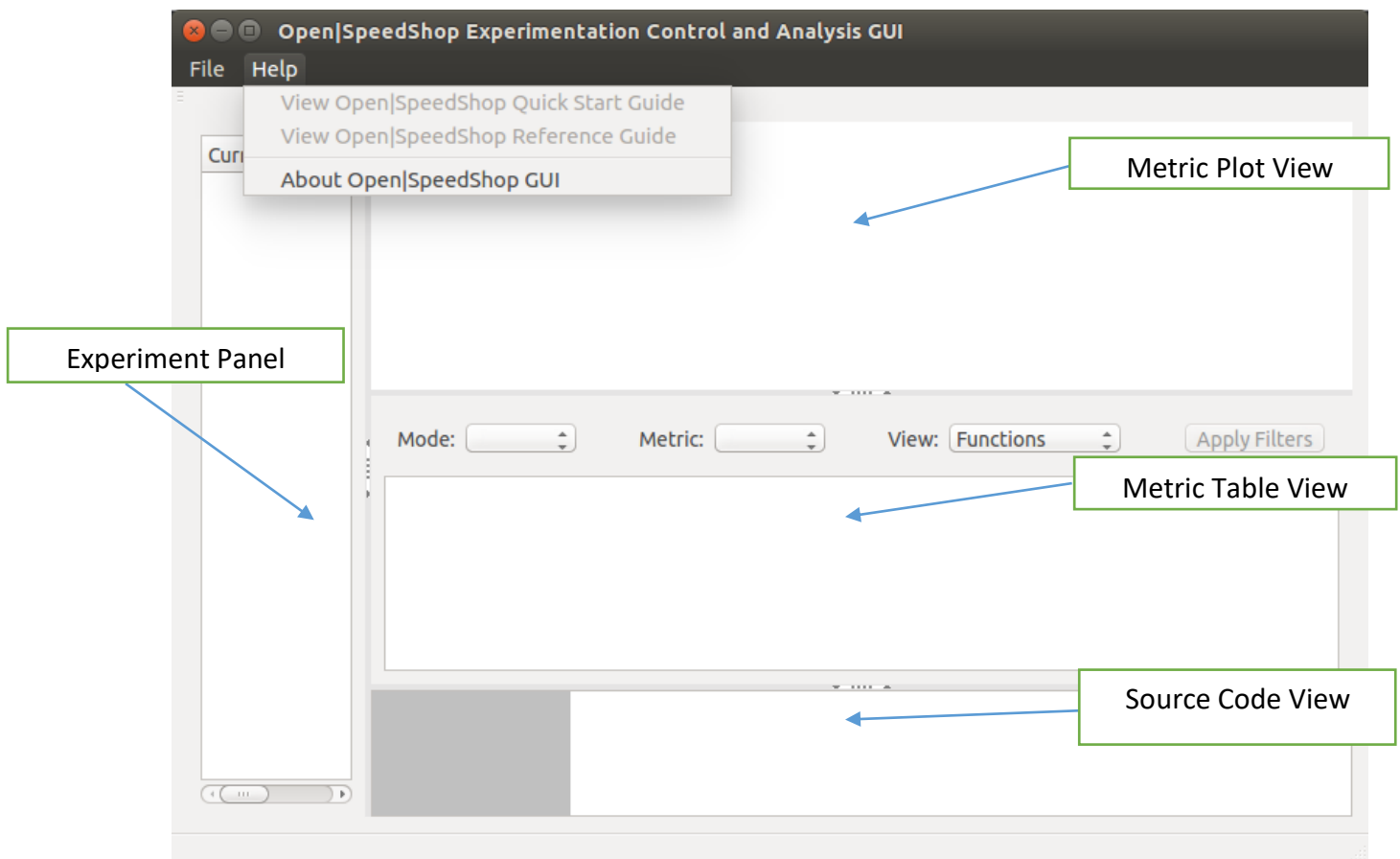


Figure 13 - Main Window User Interface Layout

The main window has a menu bar with two menu items - “File” and “Help”. Figure 1 shows the “Help” menu items which are:

- View O|SS Quick Start Guide
- View O|SS Reference Guide
- About O|SS GUI

If the O|SS Quick Start and Reference guides were installed in the standard location the menu items will be activated. The standard locations respectively are:

- \$OSS_CBTF_ROOT/\$USER_GUIDE_PATH/OpenSpeedShop_Quick_Start_Guide.pdf
- \$OSS_CBTF_ROOT/\$USER_GUIDE_PATH/OpenSpeedShop_Reference_Guide.pdf

Where:

\$USER_GUIDE_PATH = “share/doc/packages/doc/users_guide” and
\$OSS_CBTF_ROOT is the root installation directory of the O|SS CBTF components.

Activating one these menu items will open the document using the system registered application for PDF files.

The “About O|SS GUI” will open the application about dialog as shown in Figure 2. The “<http://www.openspeedshop.org>” hyperlink can be clicked to launch the system registered web browser which will automatically open the O|SS website home page.



Figure 14 - About O|SS GUI dialog

Figure 3 shows the “File” menu items “Load O|SS Experiment” and “Unload O|SS Experiment”.

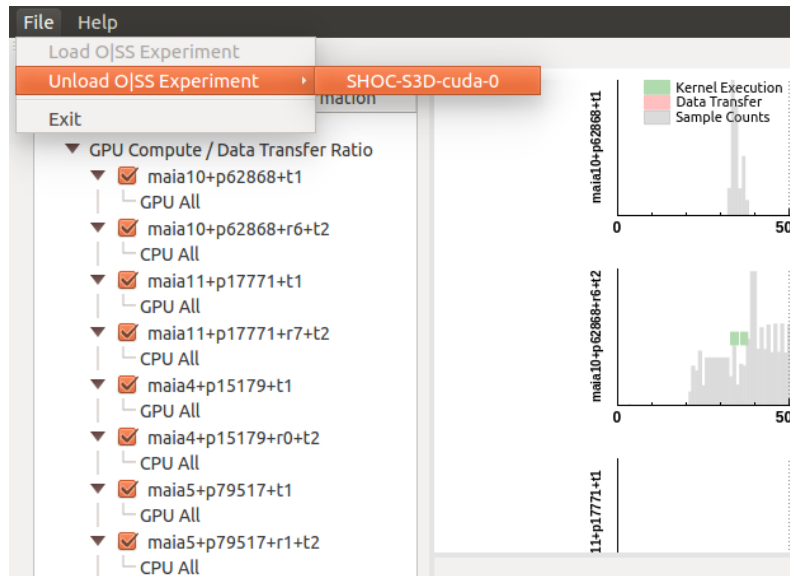


Figure 15 - File menu

Experiment loading and unloading is accomplished using the menu items under the “File” menu. The “File->Load O|SS Experiment” menu item will present an “Open File” dialog in which the user can select an O|SS experiment database to load into the application. Once an experiment has been loaded it will be added as a menu item of the “File->Unload O|SS Experiment” menu (ref Figure 3). Upon selection of the experiment in the “File->Unload O|SS Experiment” menu the user will be presented with a dialog to confirm the users desire to unload the experiment (ref. Figure 4).

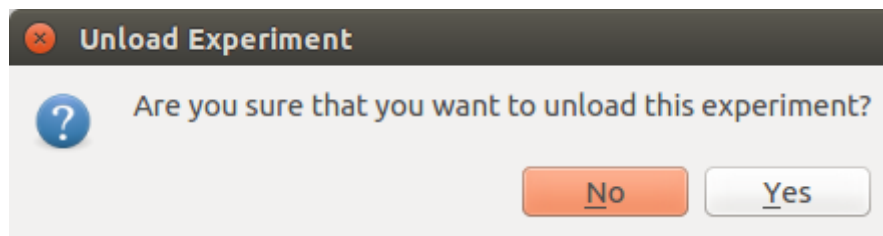


Figure 16 - Confirm Unload Experiment Dialog

The Experiment Panel is on the left-hand side of the main window. Inside the Experiment Panel is the section labeled “Currently Loaded Experiment Information” (ref. Figure 5). For the experiment that is currently loaded, this section shows the name of the loaded experiment (without the “.openss” file extension) at the top level of a tree view providing details regarding the application process identifying each parallel thread of execution. For CUDA experiments this information is shown under the tree view level titled “GPU Compute / Data Transfer Ratio”; otherwise this tree view level is titled “Thread Groups”. Currently, only one experiment can be loaded at a time. If another experiment is desired to be analyzed, then the user

needs to unload the current experiment from the application before loading another.

Each of the parallel executions (processes, threads, ranks, GPUs), called “components” in O|SS, are listed under the “GPU Compute / Data Transfer Ratio” or “Thread Groups” item in the following format:

<hostname>--<process id>--<rank> OR <hostname>--<process id>--<rank>--<thread id>

Where:

- <hostname> is the name of the computer (with domain name removed)
- <process id> is the UNIX process id
- <rank> is the MPI rank number (if the application is an MPI program)
- <thread id> is an unique O|SS GUI thread id for each POSIX thread id

The checkbox to the left of the component name is used to select which components will be included in the performance views on the right-hand side of the main window. Upon initial load of the experiment all components are selected. Thus, any metric views in the Metric Table View will include all components in the computations.



Figure 17 - Experiment Panel

Under the component items is another subtree level enumerating which sample counters were configured during experiment collection.

The right-hand side of the main window has the three sections providing the user detailed information collected by the experiment collector. The upper section is the Metric Plot View which provides graphical views of the metric data, including: event timelines, line graphs and bar graphs. The middle section is the Metric Table View and is where performance information is displayed in table views. The user can control the type of information displayed in the Metric Table View by using three different combo boxes labelled "Mode", "Metric" and "View". The "Mode" combo-box allows the user to select the metric view mode. The following modes have been implemented which match the OJSS CLI commands to provide basic metric information: load balance, calltree, metric comparisons for selected threads, processes, ranks or hosts and detailed event trace listings. The available mode options are: "Metric", "Load Balance", "CallTree", "Compare", "Compare By Process", "Compare By Rank", "Compare By Host", "Trace" and "Details". The mode options available for a particular experiment type varies according to the collector type – ie

sampling or trace. For example, sampling experiments such as “hwc”, “hwctime” and “hwcsamp” do not provide event traces. Thus, the “Trace” or “Details” mode option would not be available. Other experiment types do not provide load balance or calltree views. Thus, the “Load Balance” or “CallTree” modes would not be available. All experiments provide the metric view mode and each experiment has a default metric view which in most cases is similar to O|SS CLI default view (as shown with the “expview” command).

Detailed event trace views are provided by the “Details” or “Trace” modes. The “Details” mode is provided only for CUDA experiments and provides detailed examination of the CUDA events, filtered by type - Kernel Executions, Data Transfers or Both (All Events). By default, the “Time (ms)” column is sorted in ascending order. The “Trace” mode provides the event trace view for all other experiments.

For the “Metric” view mode, the user is able to view metric information, including: time, percentage, defining location, thread minimum, thread maximum and thread average. The metric type shown is selected using the “Metric” combo-box and the metric view can be changed with the “View” combo-box.

An analogy on how the Metric Table Views correlate to the O|SS CLI may be useful. The “Metric” option in the “Mode” combo-box provides information obtained from the O|SS CLI “expview” command. Within the “Metric” combo-box are a subset of the metrics that can be selected using the O|SS CLI “expview -m” command option; whereas the “View” combo-box are views selectable using the O|SS CLI “expview -v” command option. Many of the metrics selectable via the “-m” option are automatically included as columns in the table view – such as thread minimum, maximum and average metric values. The “Compare By Host”, “Compare By Process” and “Compare By Rank” options in the “Mode” combo-box provides information obtained by the O|SS CLI “expcompare” command. The O|SS CLI “expcompare” command has “-h”, “-p”, and “-r” options to specify which hosts, processes or ranks to compare and correlate to the “Compare By Host”, “Compare By Process” and “Compare By Rank” selections. The O|SS CLI has no ability to compare components which is possible in the O|SS GUI using the “Compare” selection. Components is a termed used by the O|SS CLI for each parallel thread of execution. Using the O|SS CLI these components can be listed using the “expstatus” command. The “expstatus” command also lists the available metrics (ref. Figure 6, “expstatus command”).

```

$ openss -cli -f ./SHOC-S3D-cuda-0.openss
openss>>[openss]: The restored experiment identifier is:  -x 1
openss>>expstatus

Experiment definition
{ # ExpId is 1, Status is Terminated, Saved database is ./SHOC-S3D-cuda-0.openss
  Performance data spans 4.904680 seconds  from 2017/04/07 14:25:09 to 2017/04/07
  14:25:14
  (none)
  Executables Involved:
    (none)
  Currently Specified Components:
    -h maia10 -p 62868 -t -1
    -h maia10 -p 62868 -t 0 -r 6
    -h maia11 -p 17771 -t -1
    -h maia11 -p 17771 -t 0 -r 7
    -h maia4 -p 15179 -t -1
    -h maia4 -p 15179 -t 0 -r 0
    -h maia5 -p 79517 -t -1
    -h maia5 -p 79517 -t 0 -r 1
    -h maia6 -p 17084 -t -1
    -h maia6 -p 17084 -t 0 -r 2
    -h maia7 -p 93435 -t -1
    -h maia7 -p 93435 -t 0 -r 3
    -h maia7 -p 93435 -t 4 -r 3
    -h maia8 -p 71904 -t -1
    -h maia8 -p 71904 -t 0 -r 4
    -h maia8 -p 71904 -t 1 -r 4
    -h maia9 -p 45031 -t -1
    -h maia9 -p 45031 -t 0 -r 5
    -h maia9 -p 45031 -t 1 -r 5
  Previously Used Data Collectors:
    cuda
  Metrics:
    cuda::count_counters
    cuda::count_exclusive_details
    cuda::exec_exclusive_details
    cuda::exec_inclusive_details
    cuda::exec_time
    cuda::periodic_samples
    cuda::xfer_exclusive_details
    cuda::xfer_inclusive_details
    cuda::xfer_time
  Parameter Values:
  Available Views:
    cuda
}

```

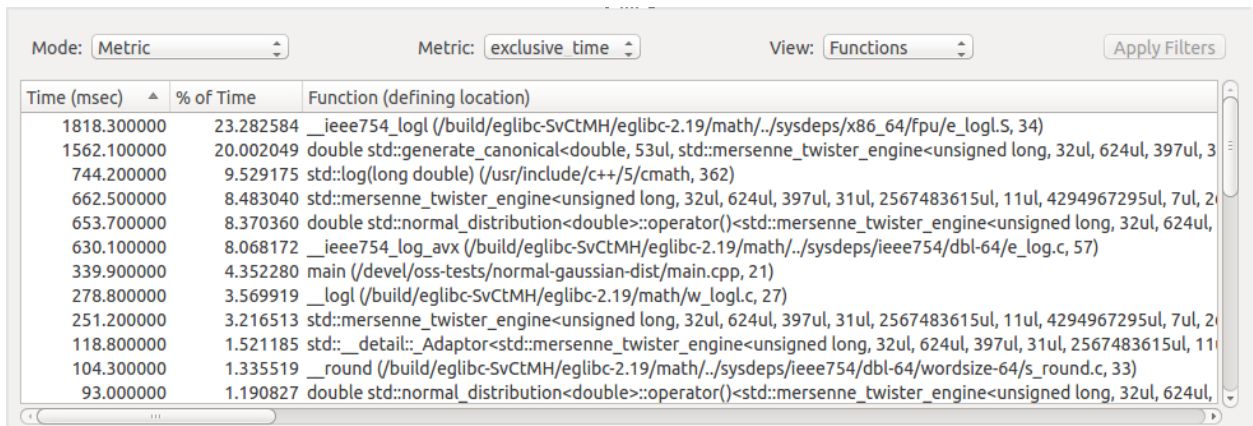
Figure 6 - expstatus command

The hosts, processes, ranks or components to compare are selected from the Experiment Panel. Based on the selections a set of unique hostnames, process identifiers or thread identifiers is generated and provides sets of performance data to use in the calculation of Metric Table View information. Currently the component selections are not used in the generation of the Metric Plot View event timelines, line graphs or bar graphs. In the near future a better means to select hostnames, rank numbers, process and thread identifiers will be implemented.

For the “Metric” view the time interval for metric computations depends on the visible range of the graph timeline and for the “Details” or “Trace” modes the time interval is used to filter which trace events are shown in the table. As the user changes the graph timeline by zooming into the graph or panning the timeline left or

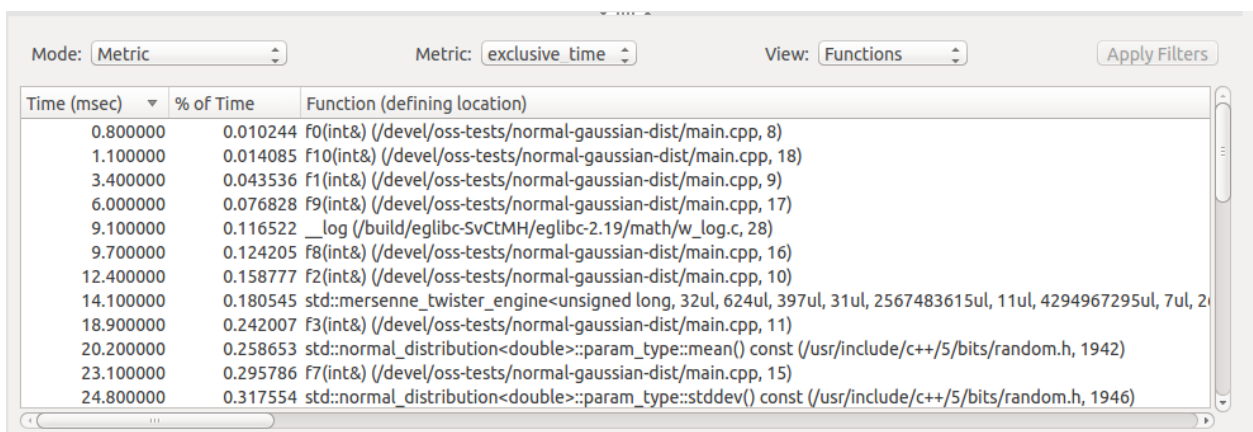
right, the Metric Table View is dynamically updated. There is a delay threshold between the time the user pauses or completes timeline changes and the actual kickoff of the processing involved for the Metric Plot or Metric Table View updates.

The row items in the table view can be ordered by clicking on a column header (ref. Figure 7 and 8) to toggle between ascending and descending order using the selected column as the key for sorting. Notice the upward and downward pointing triangle icons in the column header being used to sort the rows in the table. The upward triangle icon represents ascending order sorting and the downward triangle icon is descending order.



Time (msec) ▲	% of Time	Function (defining location)
1818.300000	23.282584	__ieee754_logl (/build/eglibc-SvCtMH/eglibc-2.19/math/./sysdeps/x86_64/fpu/e_logl.S, 34)
1562.100000	20.002049	double std::generate_canonical<double, 53ul, std::mersenne_twister_engine<unsigned long, 32ul, 624ul, 397ul, 31ul, 2567483615ul, 11ul, 4294967295ul, 7ul, 2147483647ul, 268435456ul, 18446744073709551615ul>>::operator()<std::mersenne_twister_engine<unsigned long, 32ul, 624ul, 397ul, 31ul, 2567483615ul, 11ul, 4294967295ul, 7ul, 2147483647ul, 268435456ul, 18446744073709551615ul>>::operator()> const (/usr/include/c++/5/random, 1942)
744.200000	9.529175	std::log(long double) (/usr/include/c++/5/cmath, 362)
662.500000	8.483040	std::mersenne_twister_engine<unsigned long, 32ul, 624ul, 397ul, 31ul, 2567483615ul, 11ul, 4294967295ul, 7ul, 2147483647ul, 268435456ul, 18446744073709551615ul>>::operator()<std::mersenne_twister_engine<unsigned long, 32ul, 624ul, 397ul, 31ul, 2567483615ul, 11ul, 4294967295ul, 7ul, 2147483647ul, 268435456ul, 18446744073709551615ul>>::operator()> const (/usr/include/c++/5/random, 1942)
653.700000	8.370360	double std::normal_distribution<double>::operator()<std::mersenne_twister_engine<unsigned long, 32ul, 624ul, 397ul, 31ul, 2567483615ul, 11ul, 4294967295ul, 7ul, 2147483647ul, 268435456ul, 18446744073709551615ul>>::operator()> const (/usr/include/c++/5/random, 1942)
630.100000	8.068172	__ieee754_log_avx (/build/eglibc-SvCtMH/eglibc-2.19/math/./sysdeps/ieee754/dbl-64/e_log.c, 57)
339.900000	4.352280	main (/devel/oss-tests/normal-gaussian-dist/main.cpp, 21)
278.800000	3.569919	__logl (/build/eglibc-SvCtMH/eglibc-2.19/math/w_logl.c, 27)
251.200000	3.216513	std::mersenne_twister_engine<unsigned long, 32ul, 624ul, 397ul, 31ul, 2567483615ul, 11ul, 4294967295ul, 7ul, 2147483647ul, 268435456ul, 18446744073709551615ul>>::operator()<std::mersenne_twister_engine<unsigned long, 32ul, 624ul, 397ul, 31ul, 2567483615ul, 11ul, 4294967295ul, 7ul, 2147483647ul, 268435456ul, 18446744073709551615ul>>::operator()> const (/usr/include/c++/5/random, 1942)
118.800000	1.521185	std::detail::_Adaptor<std::mersenne_twister_engine<unsigned long, 32ul, 624ul, 397ul, 31ul, 2567483615ul, 11ul, 4294967295ul, 7ul, 2147483647ul, 268435456ul, 18446744073709551615ul>>::operator()> const (/usr/include/c++/5/random, 1942)
104.300000	1.335519	__round (/build/eglibc-SvCtMH/eglibc-2.19/math/./sysdeps/ieee754/dbl-64/wordsize-64/s_round.c, 33)
93.000000	1.190827	double std::normal_distribution<double>::operator()<std::mersenne_twister_engine<unsigned long, 32ul, 624ul, 397ul, 31ul, 2567483615ul, 11ul, 4294967295ul, 7ul, 2147483647ul, 268435456ul, 18446744073709551615ul>>::operator()> const (/usr/include/c++/5/random, 1942)

Figure 7 - Column sorting via mouse clicks on column header (ascending order)

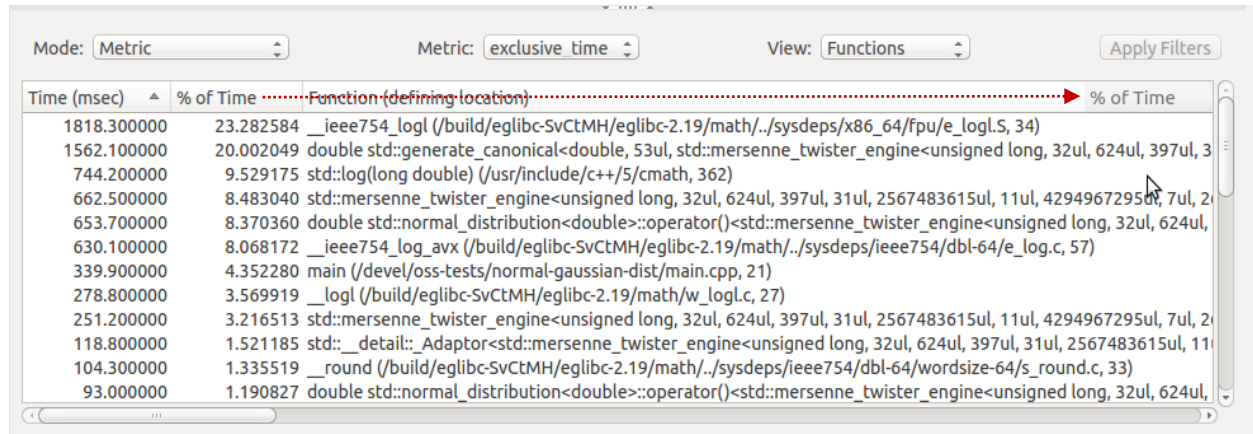


Time (msec) ▼	% of Time	Function (defining location)
0.800000	0.010244	f0(int&) (/devel/oss-tests/normal-gaussian-dist/main.cpp, 8)
1.100000	0.014085	f10(int&) (/devel/oss-tests/normal-gaussian-dist/main.cpp, 18)
3.400000	0.043536	f1(int&) (/devel/oss-tests/normal-gaussian-dist/main.cpp, 9)
6.000000	0.076828	f9(int&) (/devel/oss-tests/normal-gaussian-dist/main.cpp, 17)
9.100000	0.116522	__log (/build/eglibc-SvCtMH/eglibc-2.19/math/w_log.c, 28)
9.700000	0.124205	f8(int&) (/devel/oss-tests/normal-gaussian-dist/main.cpp, 16)
12.400000	0.158777	f2(int&) (/devel/oss-tests/normal-gaussian-dist/main.cpp, 10)
14.100000	0.180545	std::mersenne_twister_engine<unsigned long, 32ul, 624ul, 397ul, 31ul, 2567483615ul, 11ul, 4294967295ul, 7ul, 2147483647ul, 268435456ul, 18446744073709551615ul>>::operator()<std::mersenne_twister_engine<unsigned long, 32ul, 624ul, 397ul, 31ul, 2567483615ul, 11ul, 4294967295ul, 7ul, 2147483647ul, 268435456ul, 18446744073709551615ul>>::operator()> const (/usr/include/c++/5/random, 1942)
18.900000	0.242007	f3(int&) (/devel/oss-tests/normal-gaussian-dist/main.cpp, 11)
20.200000	0.258653	std::normal_distribution<double>::param_type::mean() const (/usr/include/c++/5/random, 1942)
23.100000	0.295786	f7(int&) (/devel/oss-tests/normal-gaussian-dist/main.cpp, 15)
24.800000	0.317554	std::normal_distribution<double>::param_type::stddev() const (/usr/include/c++/5/random, 1946)

Figure 8 - Column sorting via mouse clicks on column header (descending order)-

The user can alter the column ordering by holding the left-mouse button when the mouse cursor is over one of the columns and dragging it into a new position (ref Figure 9, “Changing Column Ordering”). Notice the dashed red line in the screenshot of Figure 9 showing the rubber-band effect as the “% of Time” column is dragged to the new location after the “Function (defining location)” column. The “%

of Time” text follows the cursor location as the user slides the mouse to the right. The text has a faded transparent look.



Time (msec)	% of Time	Function (defining location)	% of Time
1818.300000	23.282584	__ieee754_logl (/build/eglibc-SvCtMH/eglibc-2.19/math/./sysdeps/x86_64/fpu/e_logl.S, 34)	
1562.100000	20.002049	double std::generate_canonical<double, 53ul, std::mersenne_twister_engine<unsigned long, 32ul, 624ul, 397ul, 3	
744.200000	9.529175	std::log(long double) (/usr/include/c++/5/cmath, 362)	
662.500000	8.483040	std::mersenne_twister_engine<unsigned long, 32ul, 624ul, 397ul, 31ul, 2567483615ul, 11ul, 4294967295ul, 7ul, 2	
653.700000	8.370360	double std::normal_distribution<double>::operator()<std::mersenne_twister_engine<unsigned long, 32ul, 624ul,	
630.100000	8.068172	__ieee754_log_avx (/build/eglibc-SvCtMH/eglibc-2.19/math/./sysdeps/ieee754/dbl-64/e_log.c, 57)	
339.900000	4.352280	main (/devel/oss-tests/normal-gaussian-dist/main.cpp, 21)	
278.800000	3.569919	__logl (/build/eglibc-SvCtMH/eglibc-2.19/math/w_logl.c, 27)	
251.200000	3.216513	std::mersenne_twister_engine<unsigned long, 32ul, 624ul, 397ul, 31ul, 2567483615ul, 11ul, 4294967295ul, 7ul, 2	
118.800000	1.521185	std::_detail::_Adaptor<std::mersenne_twister_engine<unsigned long, 32ul, 624ul, 397ul, 31ul, 2567483615ul, 11	
104.300000	1.335519	__round (/build/eglibc-SvCtMH/eglibc-2.19/math/./sysdeps/ieee754/dbl-64/wordsize-64/s_round.c, 33)	
93.000000	1.190827	double std::normal_distribution<double>::operator()<std::mersenne_twister_engine<unsigned long, 32ul, 624ul,	

Figure 9 - Changing Column Ordering

The lower section on the right-hand side is the Source Code View. When the user has activated the “Metric” mode of the Metric Table View, any selections of a row in the table under the “Function (defining location)” column cause the corresponding line of the source code in the Source Code View to be displayed. Updates to the Source Code View is possible in either the “Functions”, “Statements” or “Loops” metric view (but not the “Linked Objects” metric view) assuming the source code is available on the host machine. If the user makes a selection in which the source-code cannot be found, then the Source-Code View will be empty.

If the source code is not physically located in the same location as when the executable was compiled (perhaps on another computer), then the user can specify the mapping between the original development machine location and the location on the local host machine. The dialog in which the mappings can be specified is activated from a context menu. The context menu is activated by holding down the right-mouse button when the cursor is over the row of interest under the “Function (defining location)” column. When the context menu appears near the location of the cursor, the user must select the “Modify Path Substitutions” menu item to activate the “Modify Path Substitutions Dialog” (ref Figure 10, “Modify Path Substitutions Dialog”).

The “Modify Path Substitutions Dialog” shows a table with two columns – the left column shows the original paths to the source code when the application was compiled and the right column shows the corresponding paths on the local host machine. When the dialog is activated a new entry in the table is created with the left column, “Original Path”, filled in from the information in the metric data.

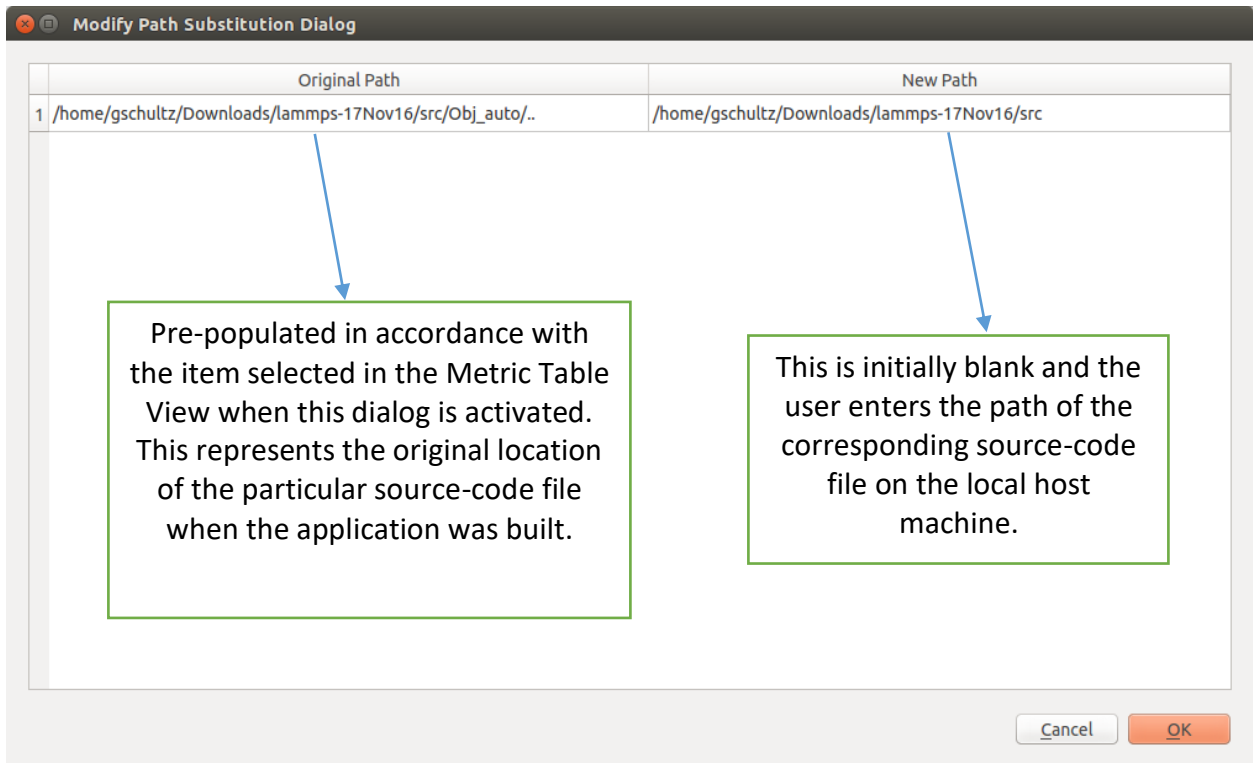


Figure 10 - Modify Path Substitutions Dialog

There are two methods in which the user can provide entry for the “New Path” item. The user can manually enter or cut-and-paste the absolute file path for the source-code into the text entry area. Alternatively there is a context-menu that can be activated by holding down the right-mouse button (ref Figure 11, “Modify Path Substitutions Dialog – Select File”). Once the context-menu appears, the user can click on the “Select File” menu item, after which the “Select Directory For File” dialog appears (ref Figure 12, “Select Directory For File Dialog”).

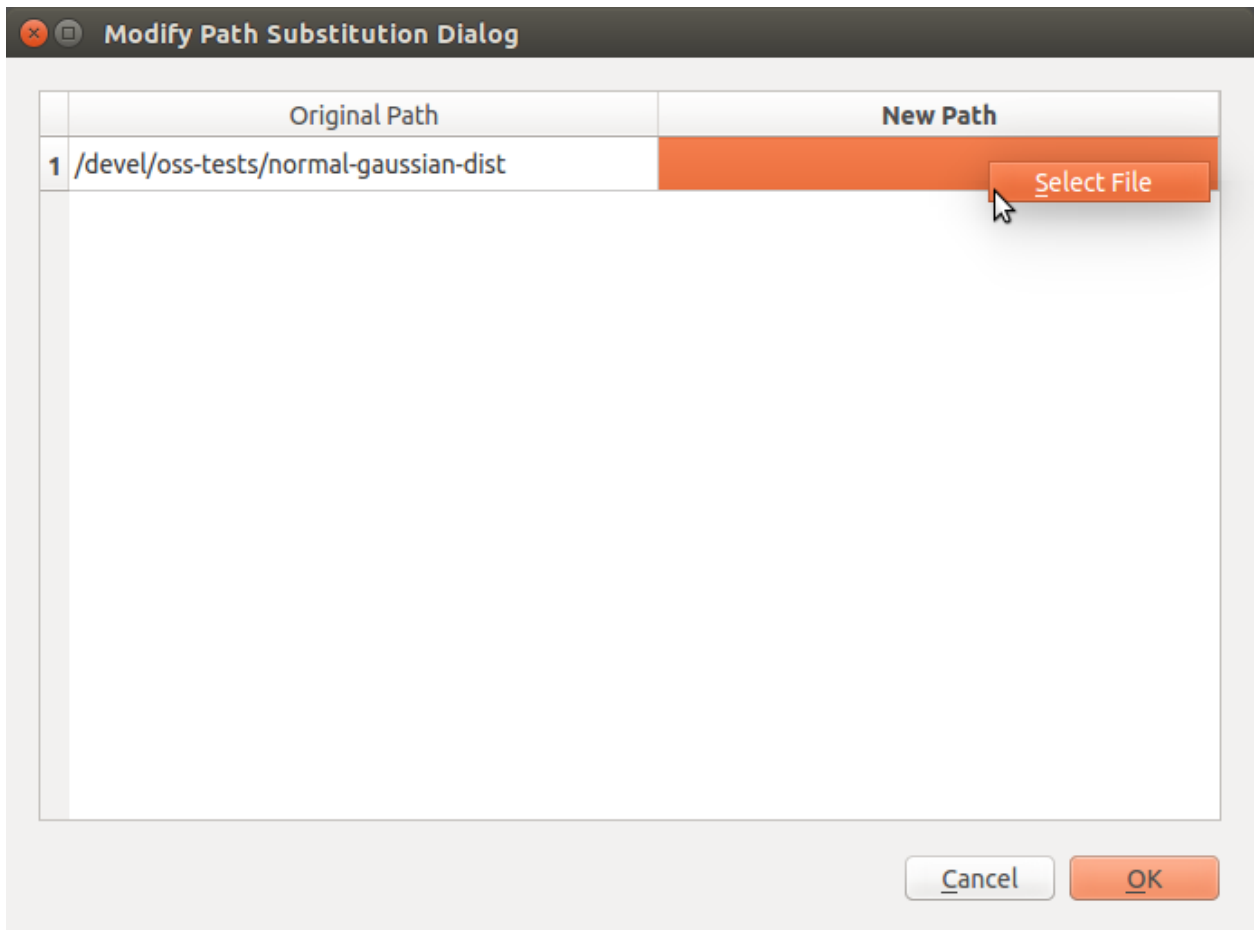


Figure 11 - Modify Path Substitutions Dialog – Select File

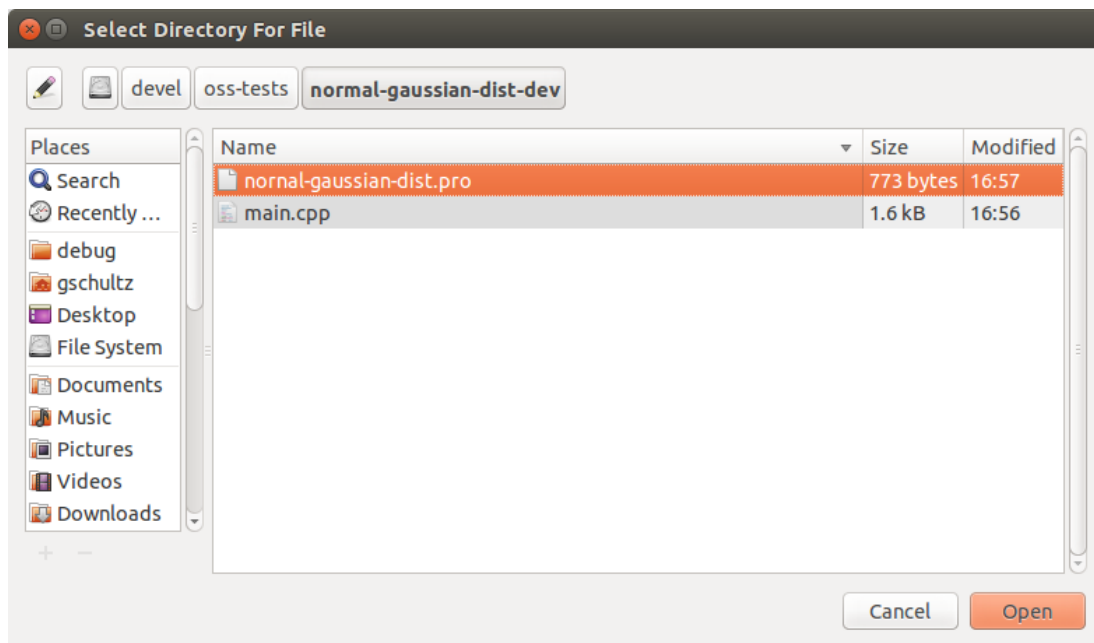


Figure 12 - Select Directory For File Dialog

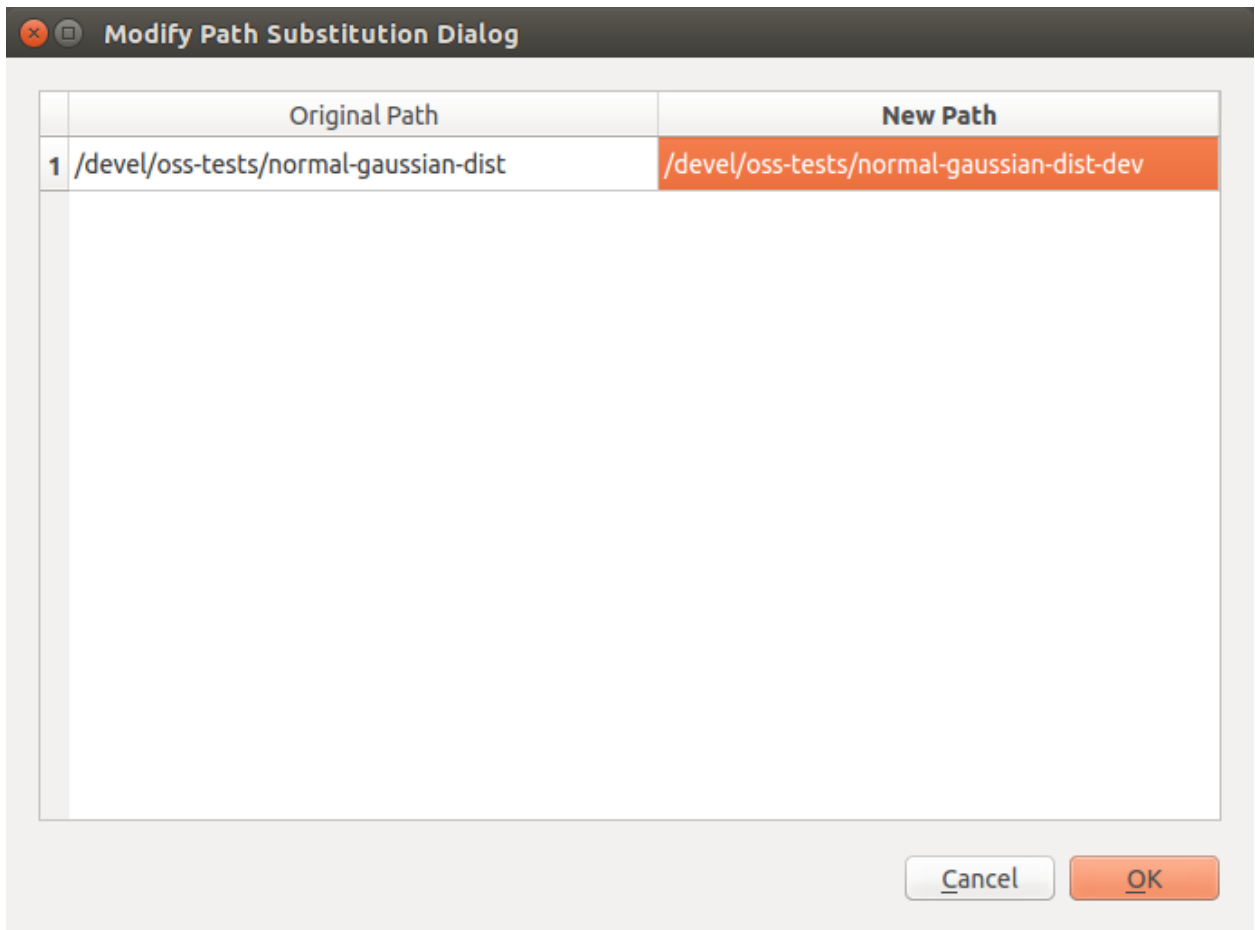
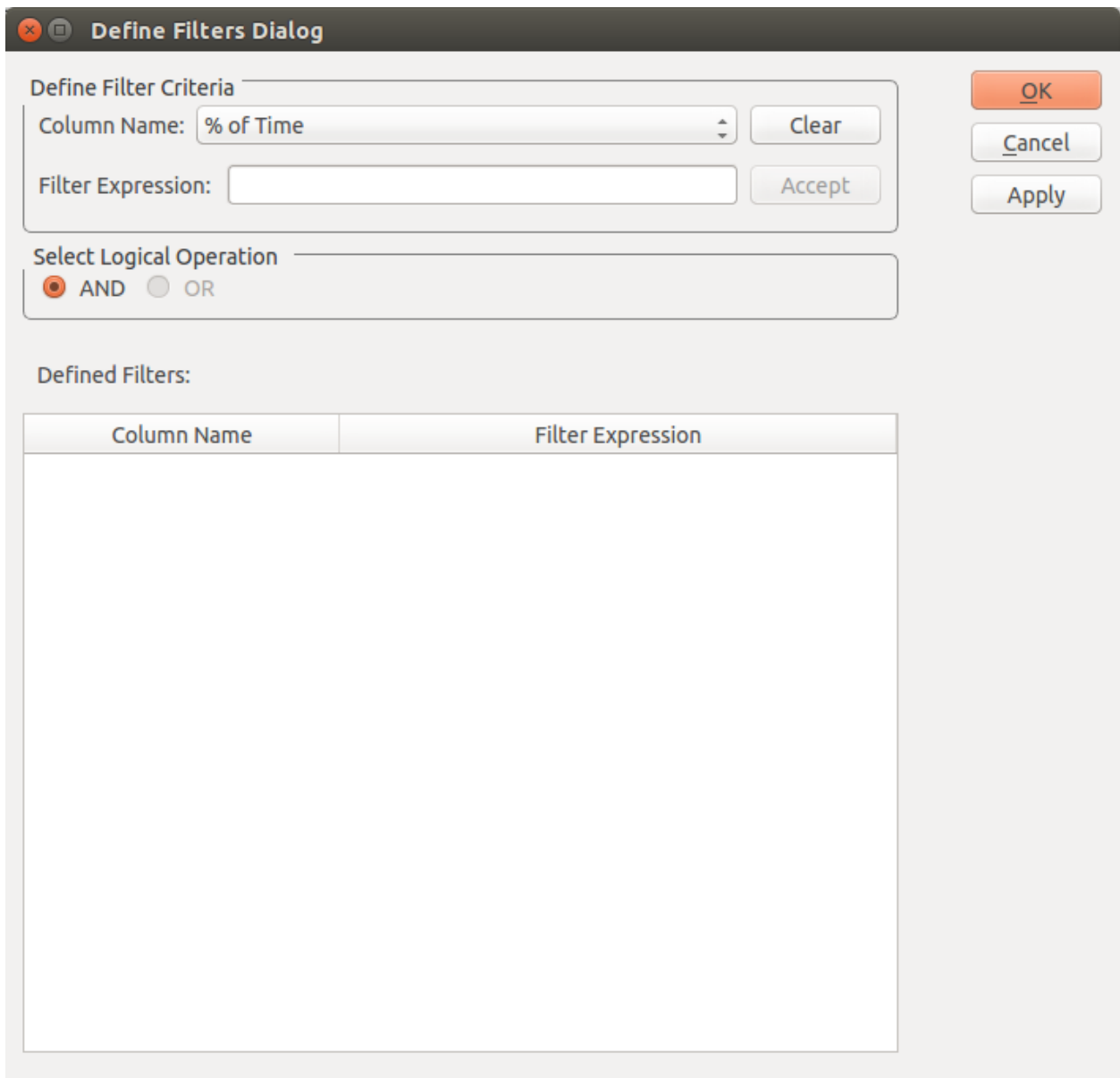


Figure 13 - Completed New Path Entry

Filtering of items shown in the Metric Table View can be achieved by using the “Define View Filters” dialog also activated via a context-menu by holding down the right-mouse button anywhere in the table area. When the context menu appears near the location of the cursor, the user must select the “Define View Filters” menu item to activate the “Define View Filters” dialog (ref Figure 14, “Define View Filters Dialog”).



The image shows a 'Define Filters Dialog' window. It has a title bar with a close button and the text 'Define Filters Dialog'. The main area is divided into sections. The first section is 'Define Filter Criteria', which contains a 'Column Name' dropdown menu with '% of Time' selected, a 'Filter Expression' text box, and buttons for 'Clear' and 'Accept'. To the right of this section are three buttons: 'OK' (orange), 'Cancel', and 'Apply'. Below the first section is a 'Select Logical Operation' section with two radio buttons: 'AND' (selected) and 'OR'. At the bottom is a 'Defined Filters' section containing a table with two columns: 'Column Name' and 'Filter Expression'. The table is currently empty.

Column Name	Filter Expression
-------------	-------------------

Figure 14 - Define View Filters Dialog

13.6.1.2 Case Studies of Using the O|SS GUI to Analyze Experiment Results

Run the O|SS GUI passing the name of the experiment database (.openss) filename using the “-f <database name>” command-line option or using the “File->Load O|SS Experiment” menu item once the application is launched.

13.6.1.2.1 Using the O|SS GUI to Analyze “pcsamp” Experiment Results

Upon loading the “pcsamp” experiment the default view appears showing a bar graph of the time distribution across the application functions attributable to the recorded PC values. These values are also shown in the Metric Table View below the bar graph (ref Figure 15, “pcsamp experiment default view”).

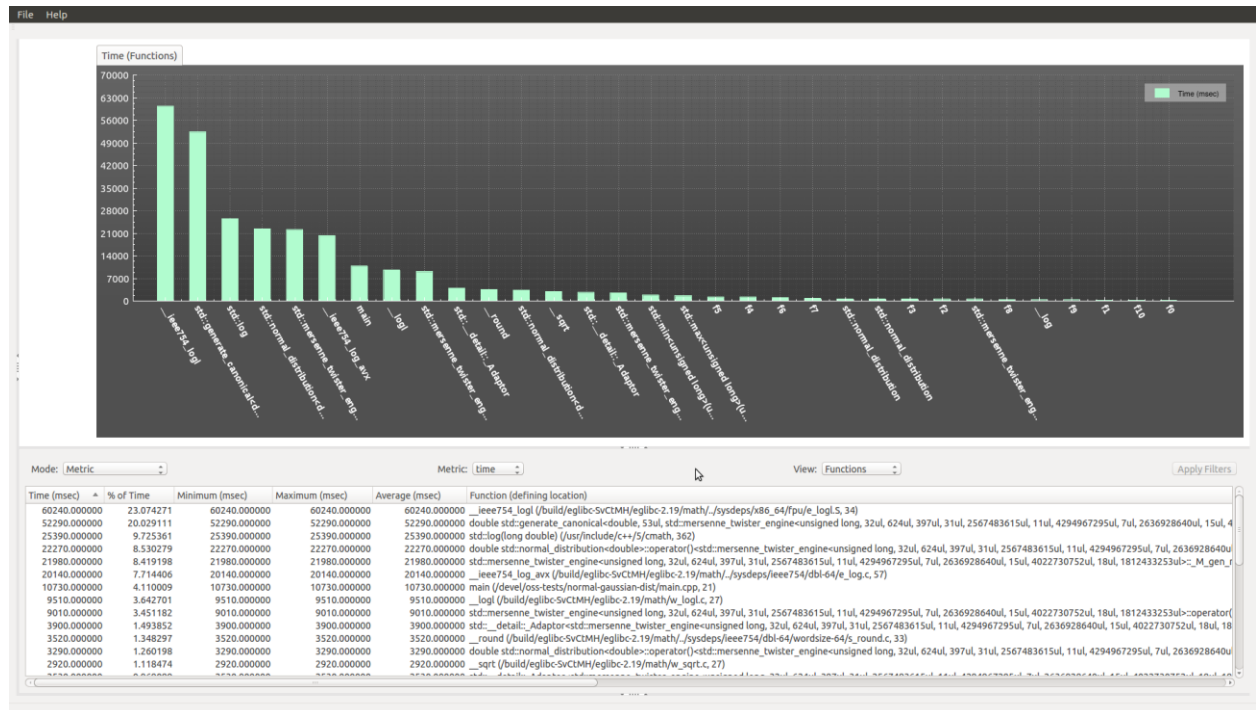


Figure 15 - pcsamp experiment default view

Generate the “Statements” view for the Metric Table View by selecting the “Statements” option in the “View” combo-box.

Define a filter for the Metric Table View to only display the statements in the “main.cpp” source-code file by activating the context-menu available by pressing the right-mouse button (ref Figure 17, “Activate Define View Filters Dialog”).

Within the “Define View Filters” dialog select “Function (defining location)” from the “Column Name” combo-box and enter “main.cpp” in the “Filter Expression” text entry area (ref Figure 18, “Define Filter for Metric Table View”). Finally, immediately apply the filter and close the “Define View Filters” dialog by pressing the “Apply” button.

Select one of the table cells under the “Function (defining location)” column to load the “main.cpp” file in the Source-Code View and center the display at the line number of the item selected. Maximize the size of the Source-Code View to examine the metric value annotations (ref Figure 19, “Source-Code View with Metric Value Annotations”).

Let's examine the source-code associated with the experiment being discussed in this example (ref Figure 16, "Source-Code Snippet - normal (Gaussian) distribution"). Lines 25 and 26 initializes the random number generator based on the Mersenne Twister algorithm. Line 32 creates an instance of the `std::normal_distribution` template class to generate random numbers of the default template parameter 'double' type according to the Normal (or Gaussian) random number distribution where the *mean* is 5 and the *standard deviation* is 2. Line 36 is a FOR loop to cause some large number of iterations to allow the periodic sampling of the collector to produce a valid statistical sample. Line 37 calls the `operator()` of the `std::normal_distribution` class to generate the next random number in the distribution and computes the nearest integer value. Based on values produces from the Normal distribution specified, the switch statement at line 38 jumps to one of the case branches at lines 39 to 49 invoking one of the `f0()` to `f10()` functions. For this Normal distribution there may be branches to the default case.

```

24      // create random number generator
25      std::random_device rd;
26      std::mt19937 mt( rd() );
27
28      // explicit normal_distribution( RealType mean = 0.0, RealType stddev = 1.0 );
29      //   where: mean      -   the  $\mu$  distribution parameter (mean)
30      //               stddev -   the  $\sigma$  distribution parameter (standard deviation)
31      //
32      std::normal_distribution<> dis( 5, 2 );
33
34      int value = 0;
35
36      for (int i=0; i<1000000000; ++i) {
37          const int randomNum = std::round( dis( mt ) );
38          switch( randomNum ) {
39              case 0: value = f0(value); break;
40              case 1: value = f1(value); break;
41              case 2: value = f2(value); break;
42              case 3: value = f3(value); break;
43              case 4: value = f4(value); break;
44              case 5: value = f5(value); break;
45              case 6: value = f6(value); break;
46              case 7: value = f7(value); break;
47              case 8: value = f8(value); break;
48              case 9: value = f9(value); break;
49              case 10: value = f10(value); break;
50              default:
51                  break;
52          }
53      }

```

Figure 16 - Source-Code Snippet - normal (Gaussian) distribution

Based on the Normal distribution specified, the results look as expected for the sampling characteristics of the collector at the time the experiment was run. The colors used as the background fill for lines of code having metric value annotations indicate the relative magnitude of time spent at the line of code. From low to high

relative magnitude the colors range from green (light and dark) to yellow (light and dark) to red (light and dark).

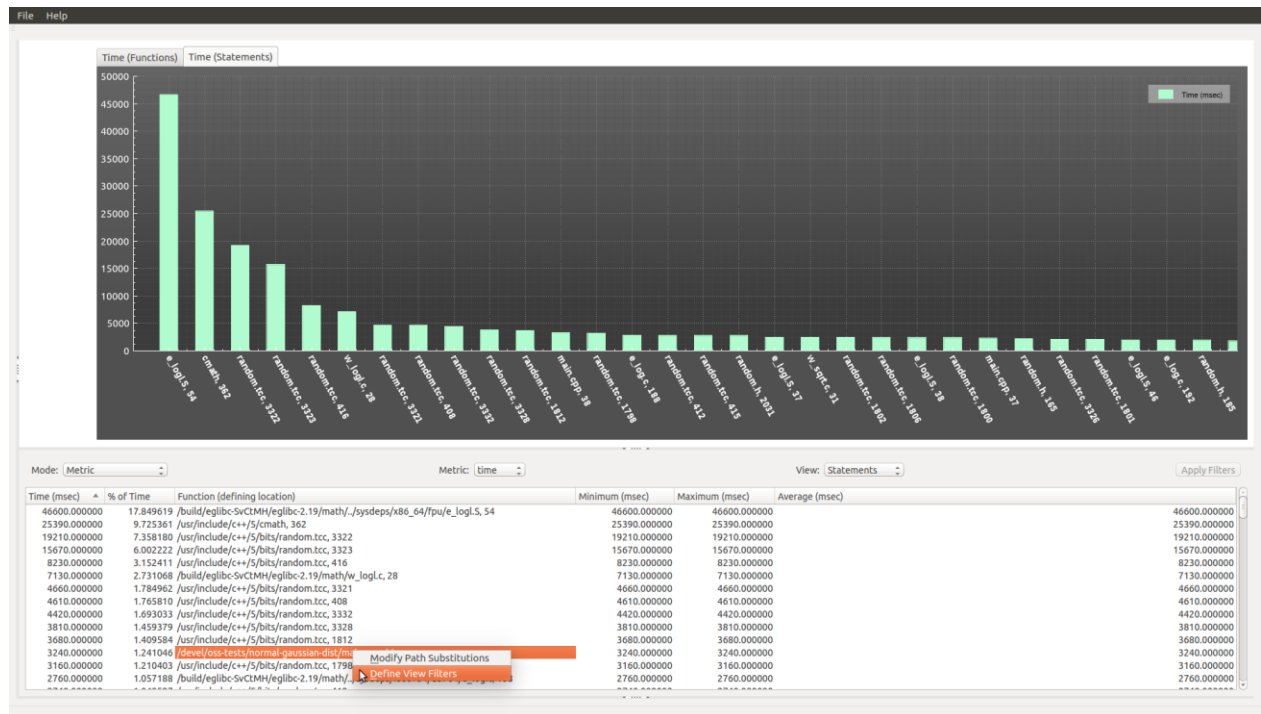


Figure 17 - Activate Define View Filters Dialog

Define Filter Criteria
Column Name: Function (defining location)
Filter Expression: main.cpp
Select Logical Operation: AND

Defined Filters:

Column Name	Filter Expression
Function (defining location)	main.cpp

Define Filter Criteria
Column Name: % of Time
Filter Expression:
Select Logical Operation: AND

Defined Filters:

Column Name	Filter Expression
Function (defining location)	main.cpp

Figure 18 - Define Filter for Metric Table View

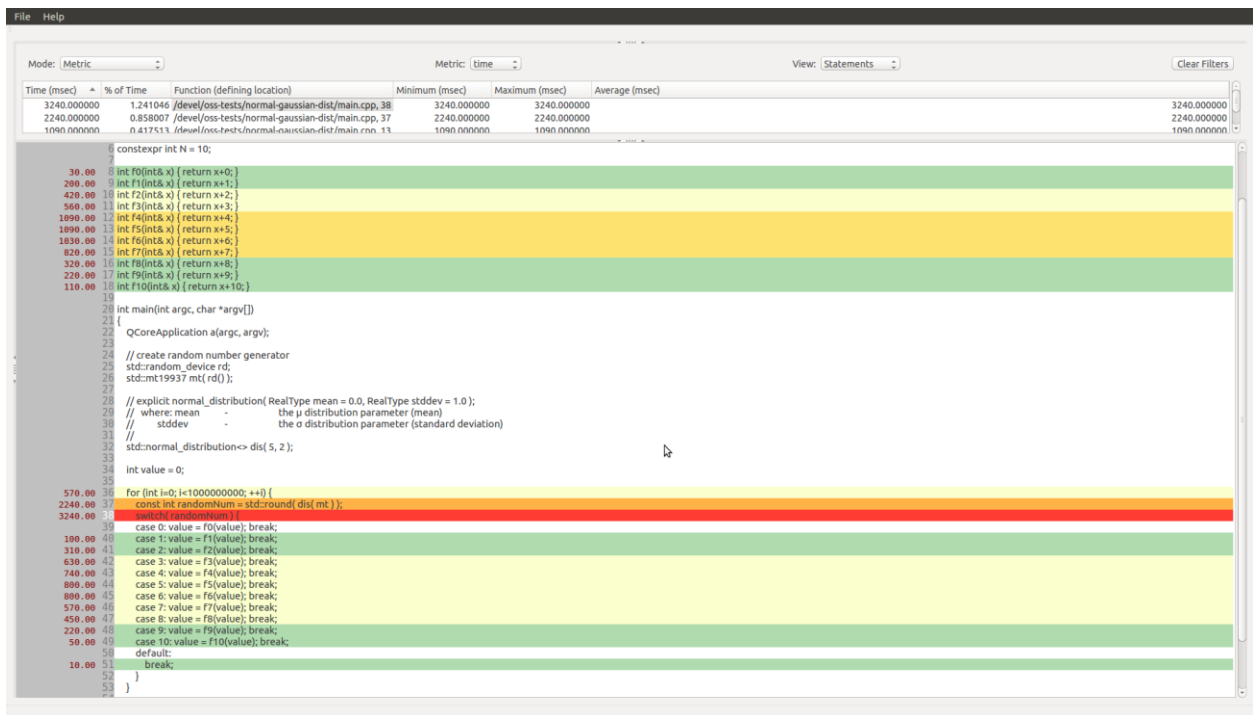


Figure 19 - Source-Code View with Metric Value Annotations

Let's look at an alternate implementation using a uniform distribution. The source-code associated with the experiment being discussed in this example (ref Figure 20, "Source-Code Snippet – uniform distribution"). Lines 24 and 25 initializes the random number generator based on the Mersenne Twister algorithm. Line 32 creates an instance of the `std::uniform_int_distribution` template class to generate random numbers of the default template parameter 'int' type over the closed interval [0, 10]. Line 34 declares a variable named 'value' and initializes to zero. Line 36 is a FOR loop to facilitate some large number of iterations to allow the periodic sampling of the collector to hopefully produce valid statistical samples. Line 37 calls the `operator()` of the `std::uniform_int_distribution` class to generate the next random number in the distribution. Since the random number should be in the closed interval [0, 10] the switch statement at line 38 should invoke one of the case branches at lines 39 to 49 invoking one of the `f0()` to `f10()` functions. The default branch should never be called in this implementation as all possible values of 'randomNum' are covered by the case branches and is provided to eliminate compiler warnings.


```

23 // create random number generator
24 std::random_device rd;
25 std::mt19937 mt( rd() );
26
27 // explicit uniform_int_distribution( IntType a = 0,
28 //                                     IntType b = std::numeric_limits<IntType>::max() );
29 //
30 // Produces random integer values i, uniformly distributed on the closed interval [a, b],
31 // that is, distributed according to the discrete probability function
32 std::uniform_int_distribution<> dis( 0, N );
33
34 int value = 0;
35
36 for (int i=0; i<1000000000; ++i) {
37     const int randomNum = dis(mt);
38     switch( randomNum ) {
39         case 0: value = f0(value); break;
40         case 1: value = f1(value); break;
41         case 2: value = f2(value); break;
42         case 3: value = f3(value); break;
43         case 4: value = f4(value); break;
44         case 5: value = f5(value); break;
45         case 6: value = f6(value); break;
46         case 7: value = f7(value); break;
47         case 8: value = f8(value); break;
48         case 9: value = f9(value); break;
49         case 10: value = f10(value); break;
50         default:
51             break;
52     }
53 }

```

Figure 20 - Source-Code Snippet - uniform distribution

Based on the uniform distribution specified, the results look as expected for the sampling characteristics of the collector at the time the experiment was run. The annotated metric values shown in the Source Code View along with the color-coded background to the source-code indicate that each case of the switch statement is executed at about the same frequency and are highlighted with the same green color (ref Figure 21, “Source-Code View with Metric Value Annotations – uniform distribution”).

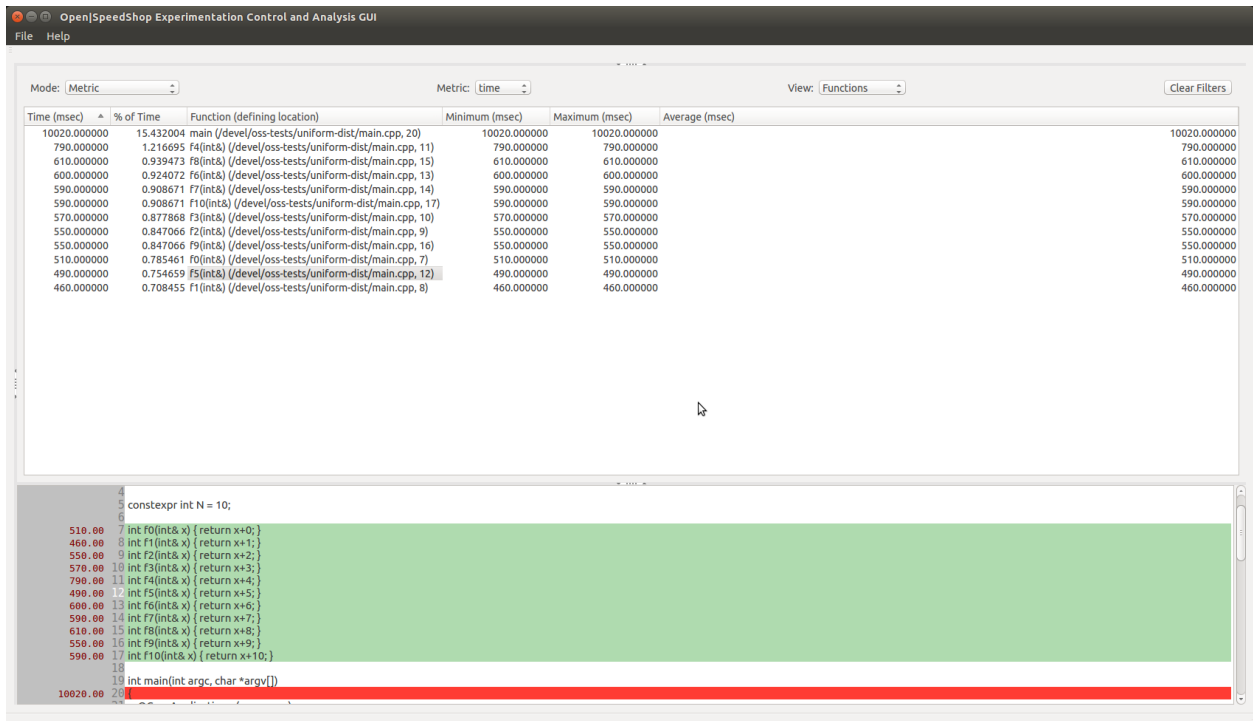


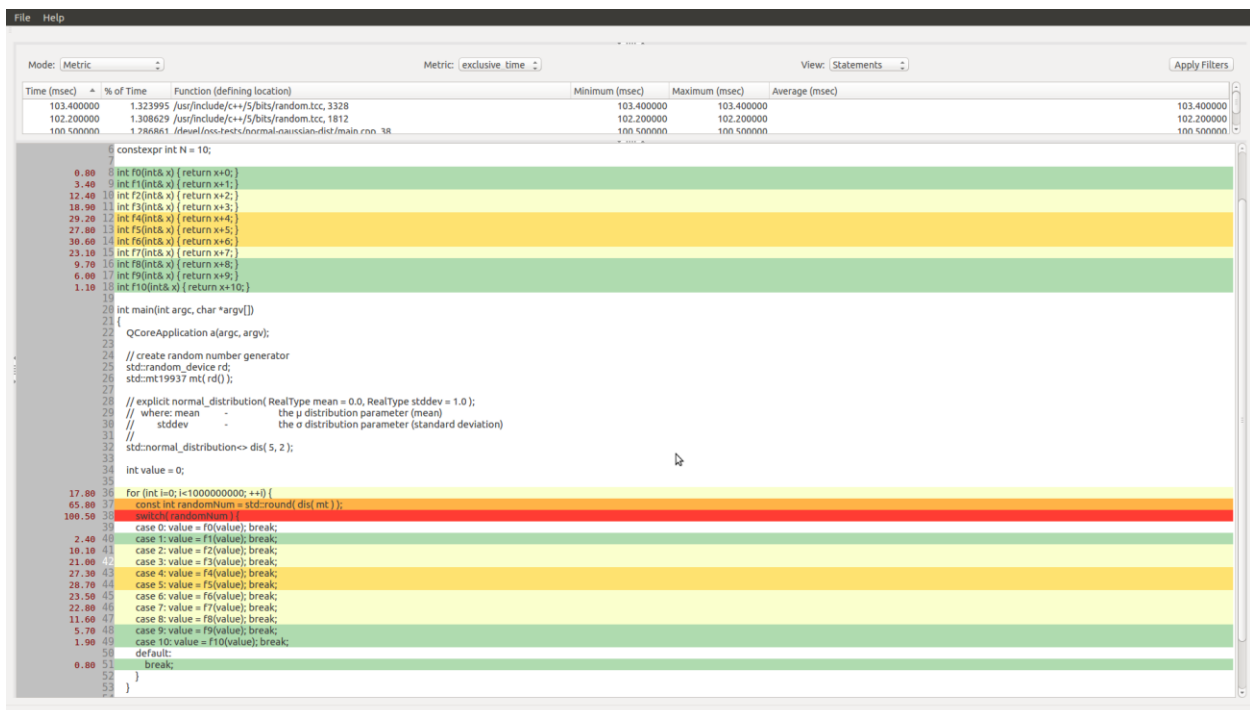
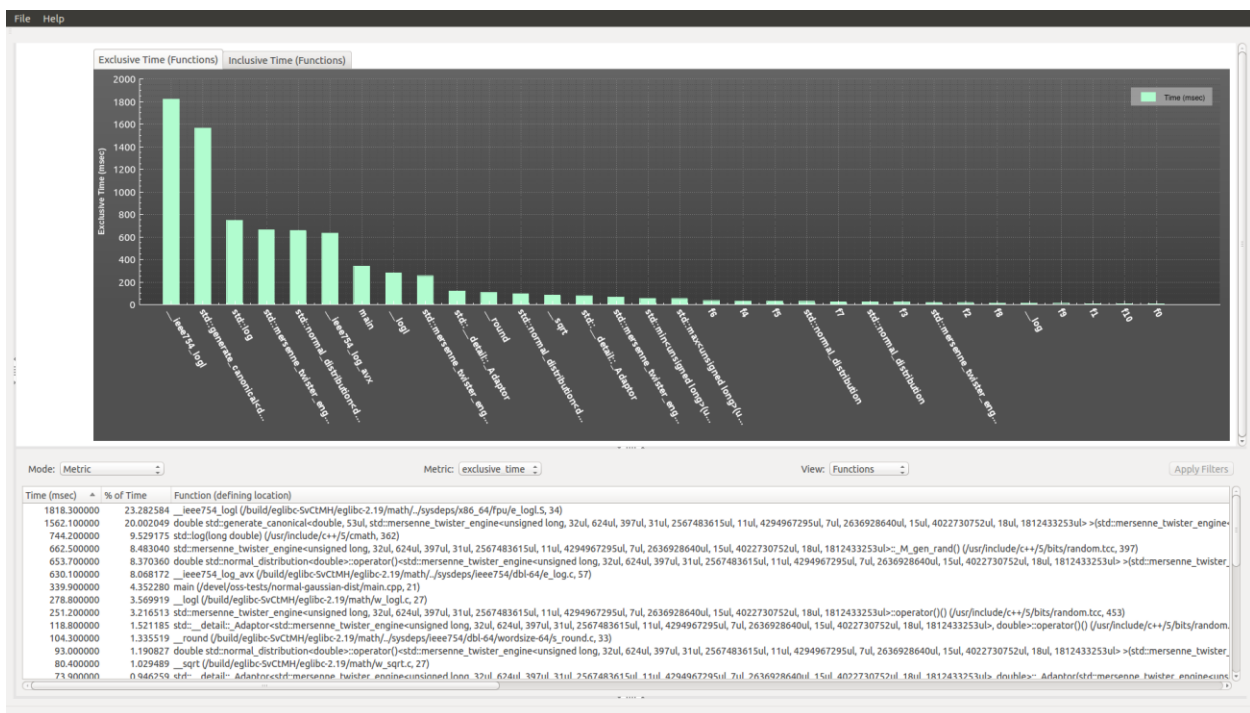
Figure 21 - Source-Code View with Metric Value Annotations – uniform distribution

13.6.1.2.2 Using the O|SS GUI to Analyze “usertime” Experiment Results

Upon loading the “usertime” experiment the default view appears showing a bar graph of the exclusive time distribution across the application functions attributable to the recorded PC values. These values are also shown in the Metric Table View below the bar graph (ref Figure 22, “usertime experiment default view”).

In order to configure the views to see just the statements within the main source file (main.cpp), configure the application similar to the steps discussed for the “pcsamp” experiment and shown in Figures 15-17 (ref. Figure 23, “Source-Code View with Metric Value Annotations (usertime)”). Since the source-code algorithm is expected to exhibit characteristics of a normal (or Gaussian) distribution having a mean of 5 and standard deviation of 2, the results look similar to those observed with the “pcsamp” experiment (ref Figure 19, “Source-Code View with Metric Value Annotations”).

Unlike the “pcsamp” experiment, the “usertime” experiment collector records call stacks for each sample. Thus, the O|SS GUI takes advantage of the availability of the call stack information to construct a calltree of all caller-callee pairs that had occurred during any particular experiment. The user can generate the calltree graph and table by selecting the “CallTree” option from the “Mode” combo-box (ref Figure 24, “Calltree Graph and Table Views” and Figure 25, “Calltree Graph Zoomed into ‘f1’ to ‘f10’ Functions”).



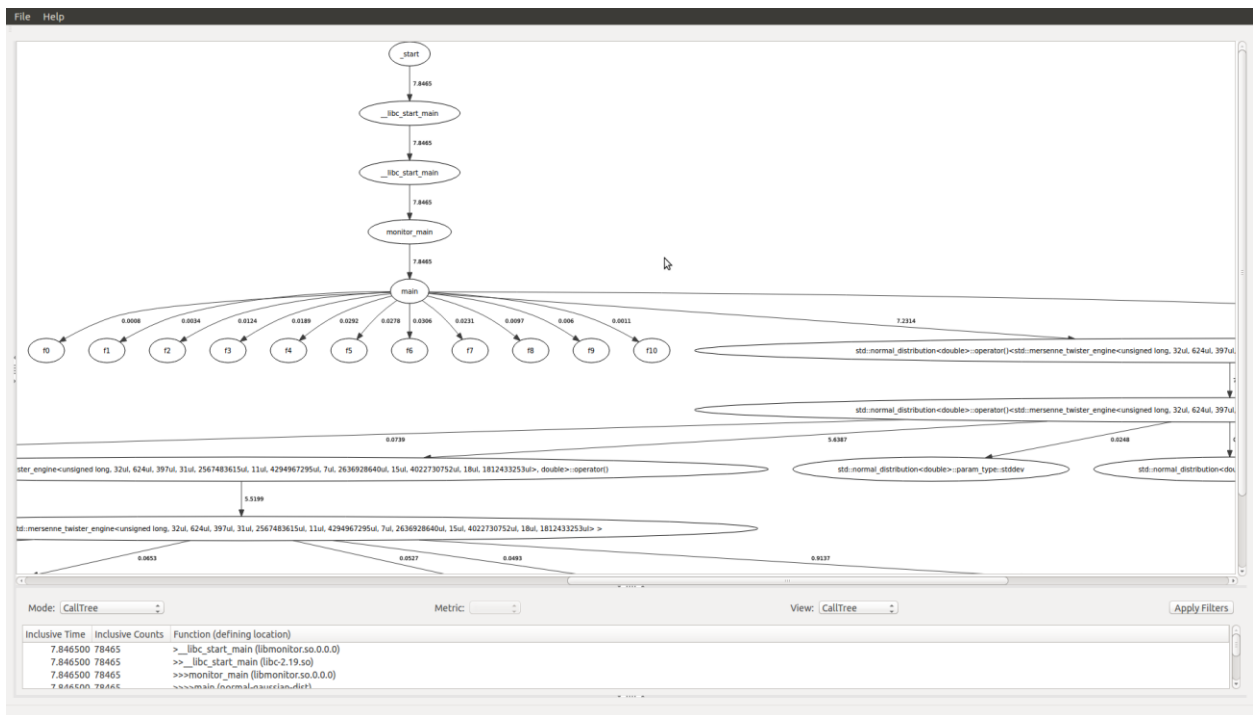


Figure 24 - Calltree Graph and Table Views

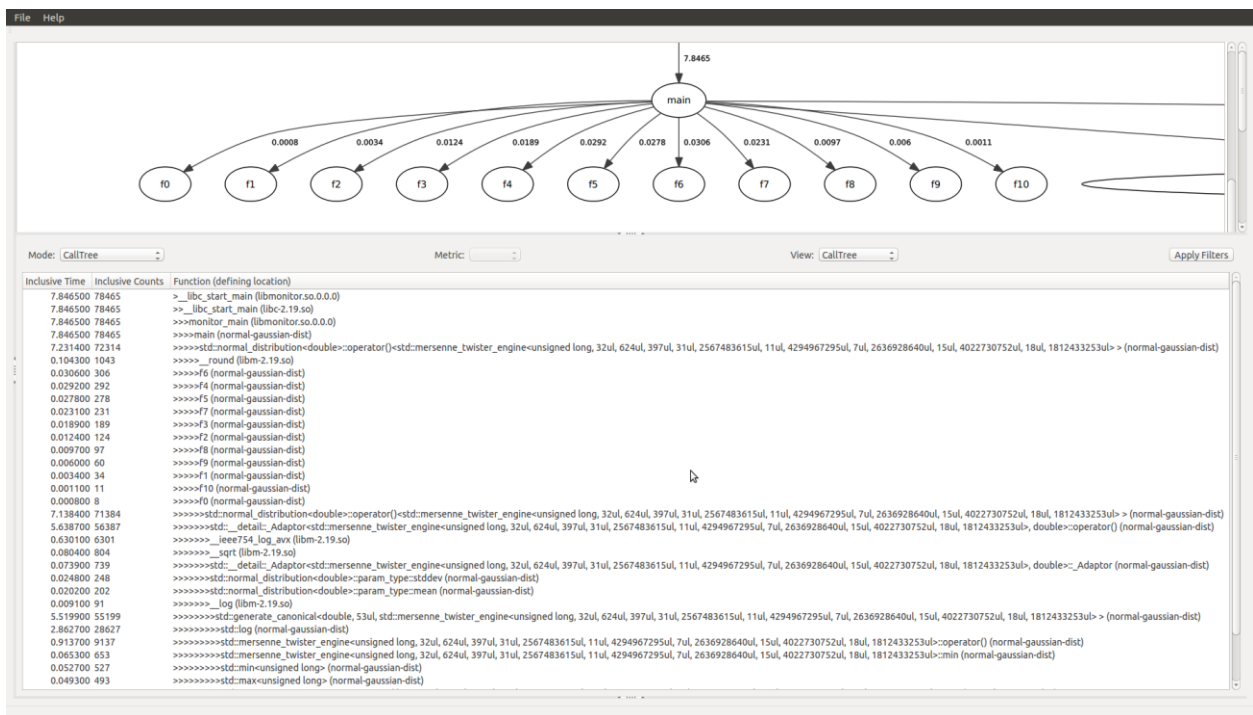


Figure 25 - Calltree Graph Zoomed into 'f1' to 'f10' Functions

13.6.1.2.3 Using the O|SS GUI to Analyze “hwc” Experiment Results

Upon loading the “hwc” experiment the default view appears showing a bar graph of the hardware counter threshold exceeded counts across the application functions attributable to the recorded PC values. These values are also shown in the Metric Table View below the bar graph (ref Figure 26, “hwc experiment default view”).

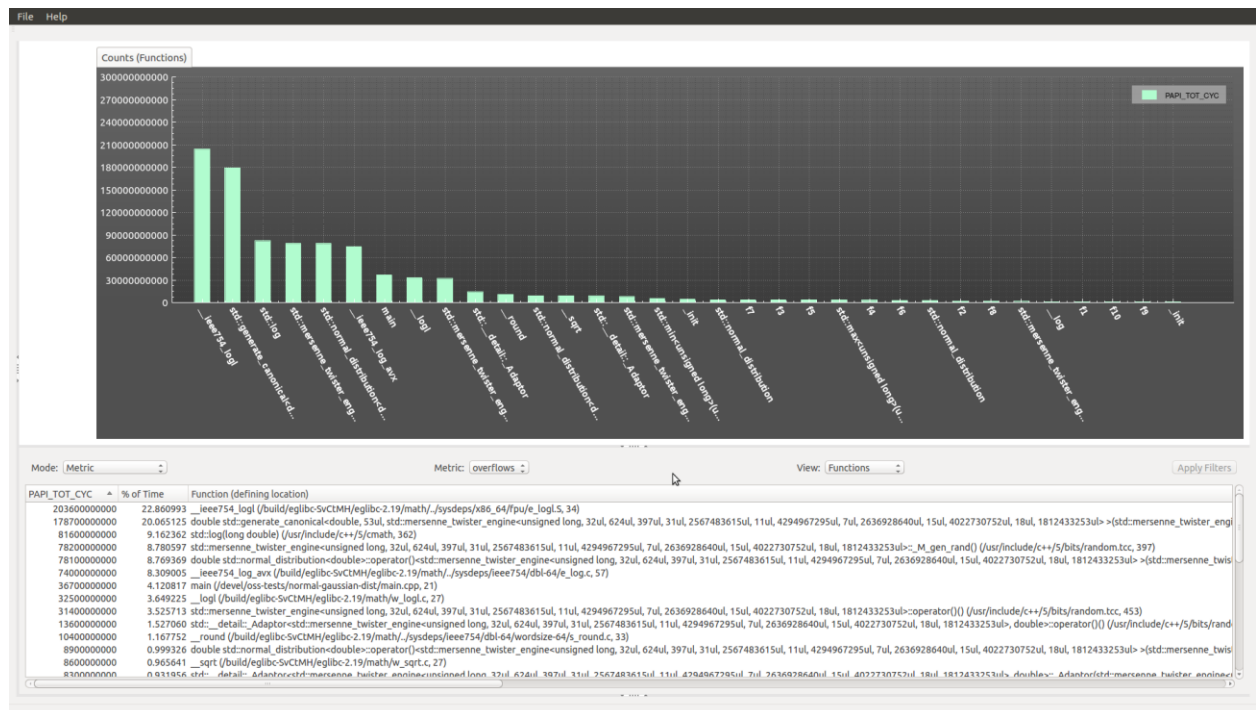


Figure 218 - hwc experiment default view

13.6.1.2.4 Using the O|SS GUI to Analyze “hwctime” Experiment Results

Upon loading the “hwctime” experiment the default view appears showing a bar graph of the hardware counter threshold exceeded counts across the application functions attributable to the recorded PC values. These values are also shown in the Metric Table View below the bar graph (ref Figure 27, “hwctime experiment default view”).

The hwctime experiment default view is a function-level view using the exclusive-time metric. In order to see a statement level view, select “Statements” in the “View” combo-box. Consequently, a new graph tab is added to the Metric Plot View titled “Exclusive Counts (Statements)” and the Metric Table View shows the metric values for the statement-level view using the exclusive-time metric (ref Figure 28, “hwctime exclusive-time metric statement-level view”). Clicking on the “Exclusive Counts (Statements)” tab shows the bar graph of the hardware counter threshold exceeded counts across the application source-code statements attributable to the

recorded PC values. If there are too many statements (or any other view type) listed in the x-axis, the labels will only show if they can be legibly displayed. Otherwise, only by scrolling the mouse-wheel to zoom into the graph area and once the labels can be displayed without overlapping will they be displayed (ref Figure 29, “zoomed graph view now showing x-axis labels”). Figure 29 has also been panned to just show the statements occurring in the main.cpp file. The label for a statement-level view is “<source-code filename>, <source-code line-number>”. So all the statements in a particular source-code file will be listed together.



Figure 219 - hwctime experiment default view

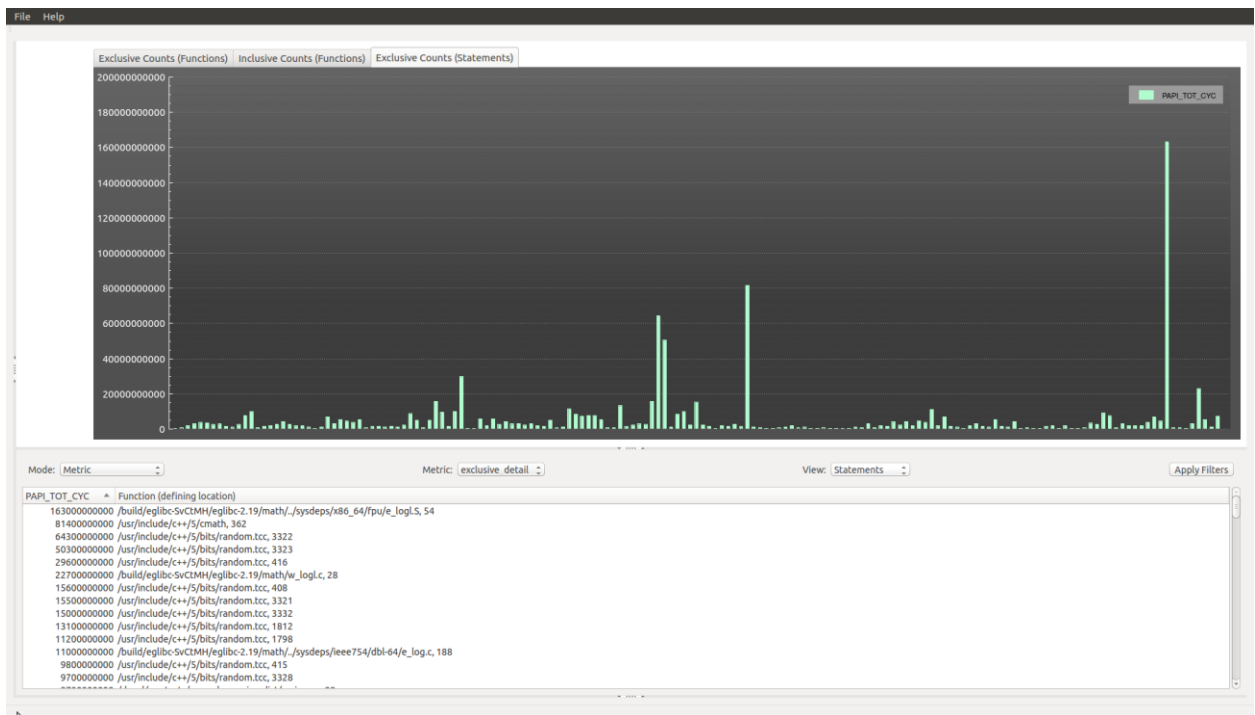


Figure 28 - hwtime exclusive-time metric statement-level view

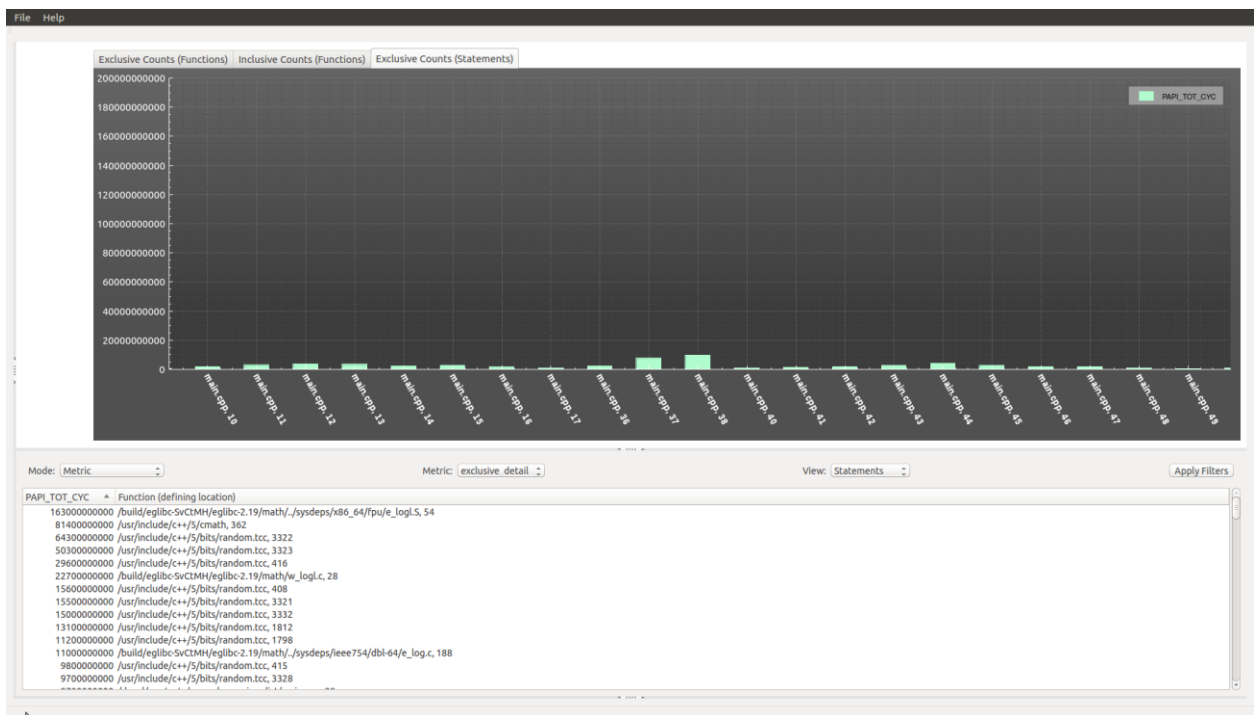


Figure 29 - zoomed graph view now showing x-axis labels

The screenshot in Figure 30, “HW counter values annotations for statement-view”

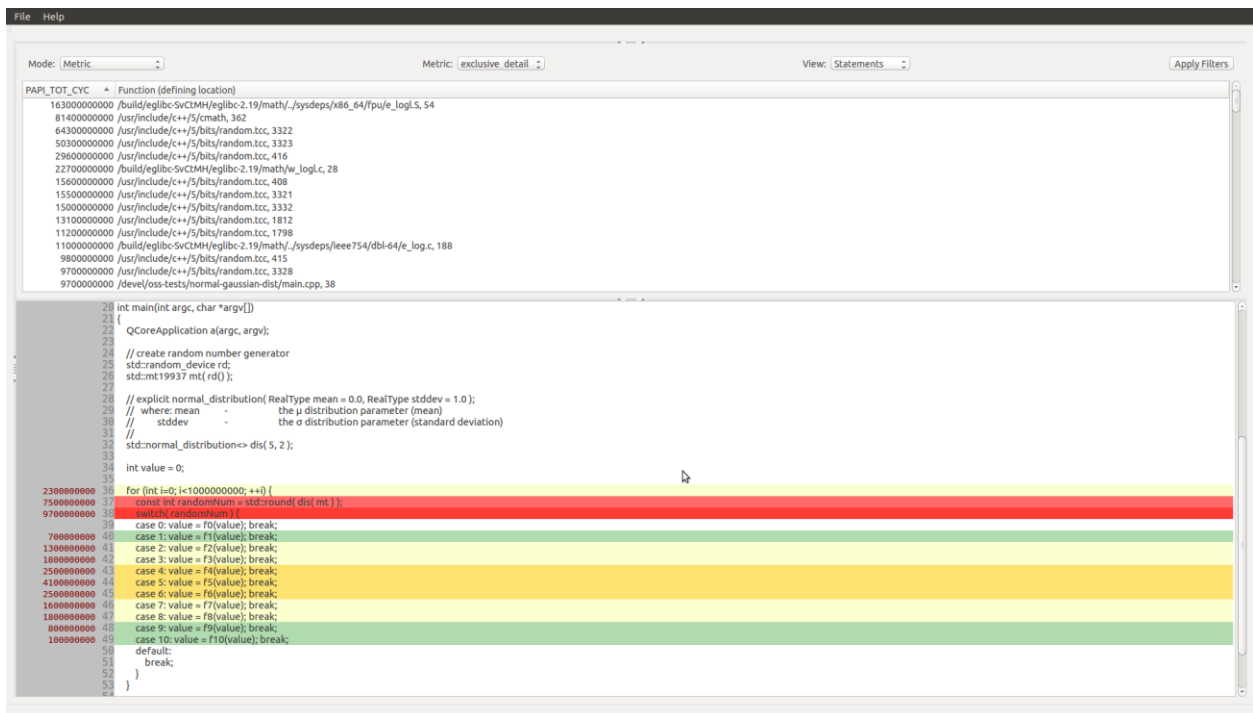


Figure 30 - HW counter values annotations for statement-view

13.6.1.2.5 Using the O|SS GUI to Analyze “hwcsamp” Experiment Results

Upon loading the “hwcsamp” experiment the default view appears showing a bar graph of the hardware counter counts across the application functions attributable to the recorded PC values. For each function there is a stacked bar graph of each of the hardware counters configured for use when the experiment was performed. These values are also shown in the Metric Table View below the bar graph (ref Figure 31, “hwcsamp experiment default view”).

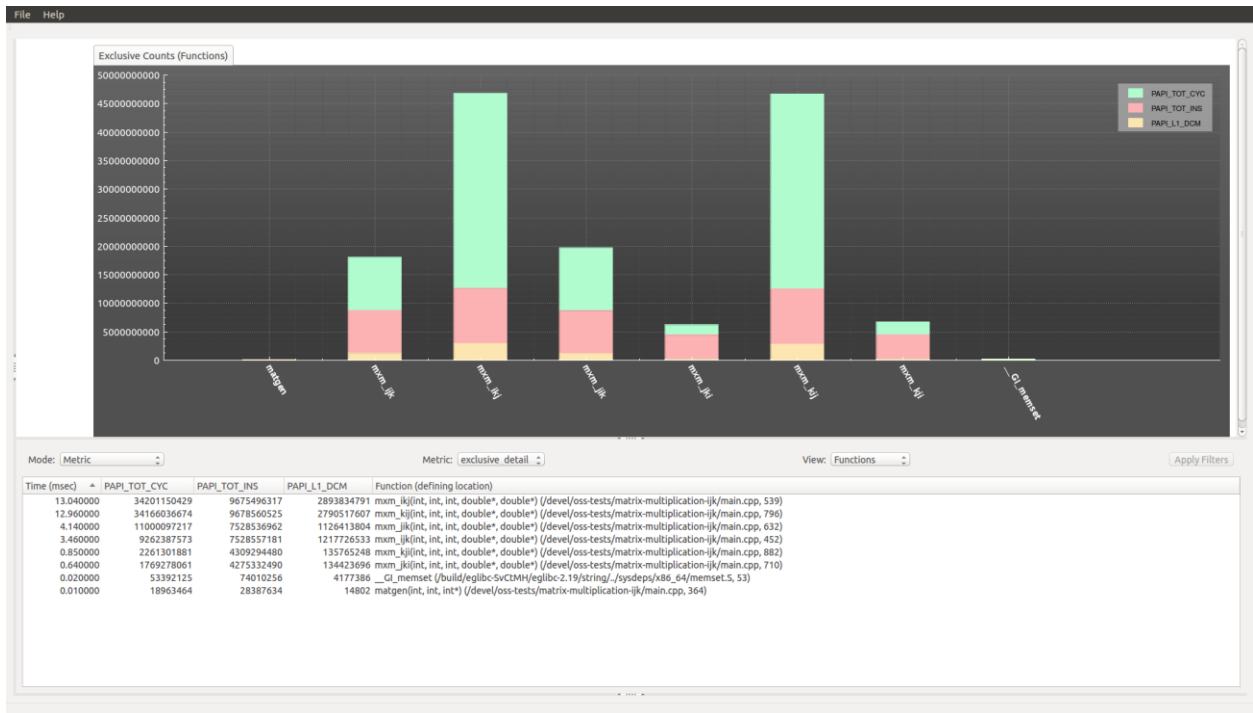


Figure 31 - hwcsamp experiment default view

The application being examined is a C++ program which performs a matrix-matrix multiplication problem $A = B * C$ using six different combinations of nested for loops:

- Nested FOR I, J, K loops
- Nested FOR I, K, J loops
- Nested FOR J, I, K loops
- Nested FOR J, K, I loops
- Nested FOR K, I, J loops
- Nested FOR K, J, I loops

The source-code for this was obtained from the following website:

http://people.sc.fsu.edu/~jburkardt/cpp_src/mxm/mxm.html

For this experiment the following three PAPI events were configured:

- PAPI_TOT_CYC - Total cycles
- PAPI_TOT_INS - Instructions issued
- PAPI_L1_DCM - Level 1 data cache misses

The output from the “osshwcsamp” experiment execution can be seen in Figure 32, “osshwcsamp experiment output”.

This example demonstrates how important it is to improve spatial and temporal locality of memory access. Proper alignment of the code and data also helps but this example doesn't demonstrate that aspect. The six nested FOR loop variants exhibit different behavior affecting the L1 cache. If the programmer can identify ways to improve L1 cache usage this also improves the usage of the other cache levels and thus application performance.

```
$ osshwcsamp "/mxm 1024 1024 1024" PAPI_TOT_CYC,PAPI_TOT_INS,PAPI_L1_DCM
[openss]: hwcsamp using default sampling rate: "100".
[openss]: hwcsamp using user specified papi events: "PAPI_TOT_CYC,PAPI_TOT_INS,PAPI_L1_DCM"
Creating topology file for frontend host eluveitie
Generated topology file: ./cbtfAutoTopology
Running hwcsamp collector.
Program: ./mxm 1024 1024 1024
Number of mrnet backends: 1
Topology file used: ./cbtfAutoTopology
executing sequential program: cbtfmun -c hwcsamp --mrnet ./mxm 1024 1024 1024
07 December 2017 07:57:01 PM

MXM:
  C++ version
  Compute matrix-matrix product A = B * C

  Matrix B is 1024 by 1024
  Matrix C is 1024 by 1024
  Matrix A will be 1024 by 1024

  Number of floating point operations = 2.14748e+09
  Estimated CPU time is 59652.3 seconds.

Method      Cpu Seconds      MegaFlopS
-----
IKJ          13.0446         164.626
IJK          3.45972        620.711
JIK          4.14068         518.63
JKI          0.646093        3323.8
KIJ          12.9603         165.697
KJI          0.856215        2508.11

MXM:
  Normal end of execution.

07 December 2017 07:57:36 PM
All Threads are finished.
default view for /devel/oss-tests/matrix-multiplication-ijk/mxm-hwcsamp-46.openss
[openss]: The restored experiment identifier is: -x 1
Performance data spans 35.128076 seconds from 2017/12/07 19:57:01 to 2017/12/07 19:57:36

Exclusive   % of CPU   papi_tot_cyc   papi_tot_ins   papi_l1_dcm Comp.   papi_tot_cyc%   Function (defining location)
CPU time    Time Intensity
in
seconds.
13.040000   37.129841   34201150429   9675496317    2893834791 0.282900       36.881472   mxm_ijk(int, int, int, double*, double*) (mxm: main.cpp,539)
12.960000   36.902050   34166036674   9678560525    2790517607 0.283280       36.843606   mxm_kij(int, int, int, double*, double*) (mxm: main.cpp,796)
4.140000    11.788155   11000097217   7528536962    1126413804 0.684406       11.862167   mxm_jik(int, int, int, double*, double*) (mxm: main.cpp,632)
3.460000    9.851936    9262387573    7528557181    1217726533 0.812810       9.988275    mxm_ijk(int, int, int, double*, double*) (mxm: main.cpp,452)
0.850000    2.420273    2261301881    4309294480    135765248 1.905670       2.438519    mxm_kji(int, int, int, double*, double*) (mxm: main.cpp,802)
0.640000    1.822323    1769278061    4275332490    134423696 2.416428       1.907935    mxm_jki(int, int, int, double*, double*) (mxm: main.cpp,710)
0.020000    0.056948    53392125      74010256      4177386 1.386164       0.057576    __GI_memset (libc-2.19.so: memset.S,53)
0.010000    0.028474    18963464      28387634      14802 1.496965    0.020450    matgen(int, int, int*) (mxm: main.cpp,364)
35.120000   100.000000  92732607424   43098175845   8302873867 0.464758       100.000000   Report Summary
```

Figure 32 - osshwcsamp experiment output

The “mxm” application calculates megaFLOPS for each of the six variants of nested FOR loop ordering. The highest megaFLOPS values computed by “mxm” are for the JKI and KJI variants. These are also the variants that show the lowest CPU cycles, instructions issued and Level 1 data cache misses (ref Figure 31, “hwcsamp experiment default view”). Figure 31 also shows the exclusive time to run each of the six FOR loop variants. The “usertime” experiment can be used to get the similar

exclusive time results (ref Figure 34, “exclusive times for nested FOR loop variants using the usertime experiment”).

Using the data collected from the PAPI_TOT_INS and PAPI_TOT_CYCLES events, the Instructions Per Cycle (IPC), also referred to as Computational Intensity (CI), can be calculated using the following formula:

$$\text{Instructions Per Cycle (IPC)} = \text{PAPI_TOT_INS} / \text{PAPI_TOT_CYCLES}$$

Using the data shown in Figure 31, “hwcsamp experiment default view” or in Figure 32, “osshwcsamp experiment output”, the Instructions Per Cycle (IPC) or Computational Intensity (CI), can be calculated. Table 1, “matrix-matrix multiplication FOR loop variant comparison”, provides a comparison of the IPC values for each of the six FOR loop variants.

Metric	IJK	IKJ	JKI	JKI	KIJ	KJI
PAPI_TOT_INS	752855718 1	9675496317	7528536962	427533249 0	9678560525	430929448 0
PAPI_TOT_CYC	926238757 3	3420115042 9	1100009721 7	176927806 1	3416603667 4	226130188 1
IPC	0.813	0.283	0.684	2.416	0.283	1.906
MFLOPS	620.711	164.626	620.711	3323.8	165.697	2508.11

Table 1- matrix-matrix multiplication FOR loop variant comparison

The source-code for the JKI and KJI variants are shown in Figure 33, “JKI / KJI variant source-code”. There has been a slight modification to the source-code provided by John Burkardt in the reference cited above. The difference is the loop to initialize the array ‘a’ to zeros has been replaced with std::fill().

<pre>double mxm_jki (int n1, int n2, int n3, double b[], double c[]) { double* a = new double[n1*n3]; std::fill(a, a+n1*n3, 0); double cpu_seconds = cpu_time(); for (j = 0; j < n3; j++) { for (k = 0; k < n2; k++) { for (i = 0; i < n1; i++) { a[i+j*n1] += (b[i+k*n1] * c[k+j*n2]); } } } cpu_seconds = cpu_time() - cpu_seconds; delete[] a; return cpu_seconds; }</pre>	<pre>double mxm_kji (int n1, int n2, int n3, double b[], double c[]) { double* a = new double[n1*n3]; std::fill(a, a+n1*n3, 0); double cpu_seconds = cpu_time(); for (k = 0; k < n2; k++) { for (j = 0; j < n3; j++) { for (i = 0; i < n1; i++) { a[i+j*n1] += (b[i+k*n1] * c[k+j*n2]); } } } cpu_seconds = cpu_time() - cpu_seconds; delete[] a; return cpu_seconds; }</pre>
---	---

Figure 33 - JKI / KJI variant source-code

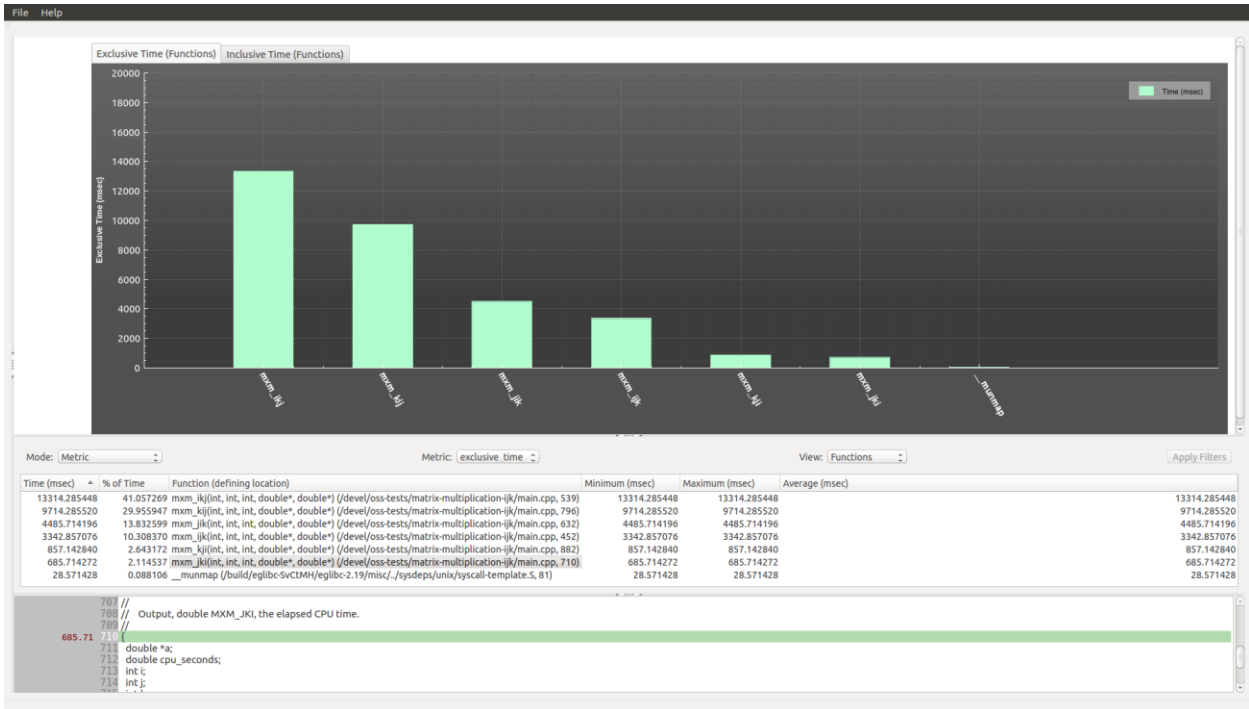


Figure 34 - exclusive times for nested FOR loop variants using the usertime experiment

13.6.1.2.6 Using the O|SS GUI to Analyze “omptp” Experiment Results

Upon loading the “omptp” experiment the default view appears showing the exclusive time metric values for the functions view (ref Figure 35, “omptp experiment default view”). Currently there is no graph generated in the Metric Plot View.

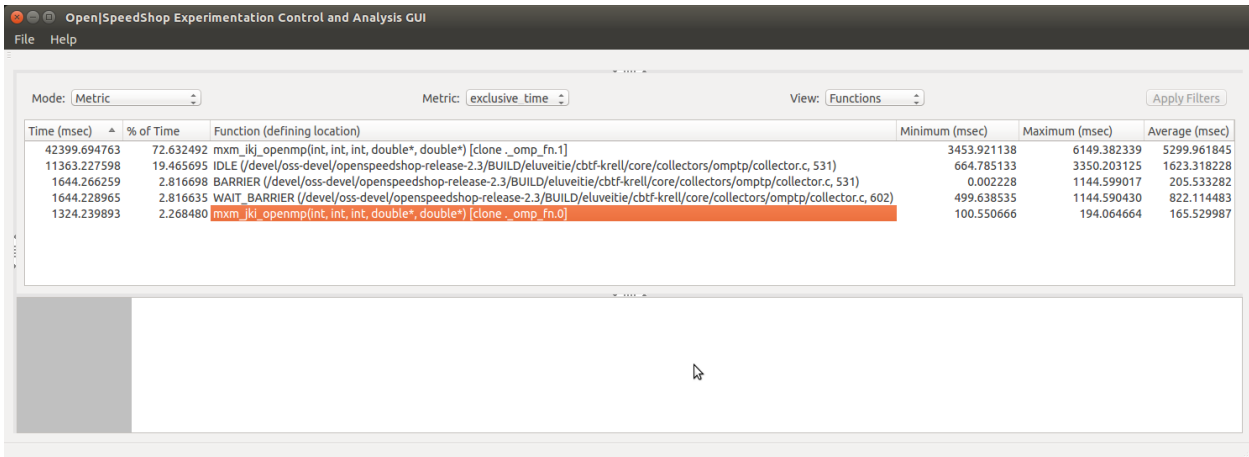


Figure 35 - omptp experiment default view

A calltree graph can be generated by selecting the “CallTree” option in the “Mode” combo-box. After processing all call stacks recorded from the experiment and stored in the database file, a directed graph is displayed in the Metric Plot View. The directed graph has functions as the nodes and the edges show the path from the caller function to callee function (ref Figure 36, “calltree graph showing caller-callee relationships”).

A load balance metric view can be generated by selecting the “Load Balance” option in the “Mode” combo-box. The load balance view shows which threads have the minimum and maximum time for each function as well as which thread is closest to the average time (ref Figure 37, “omptp load balance view”).

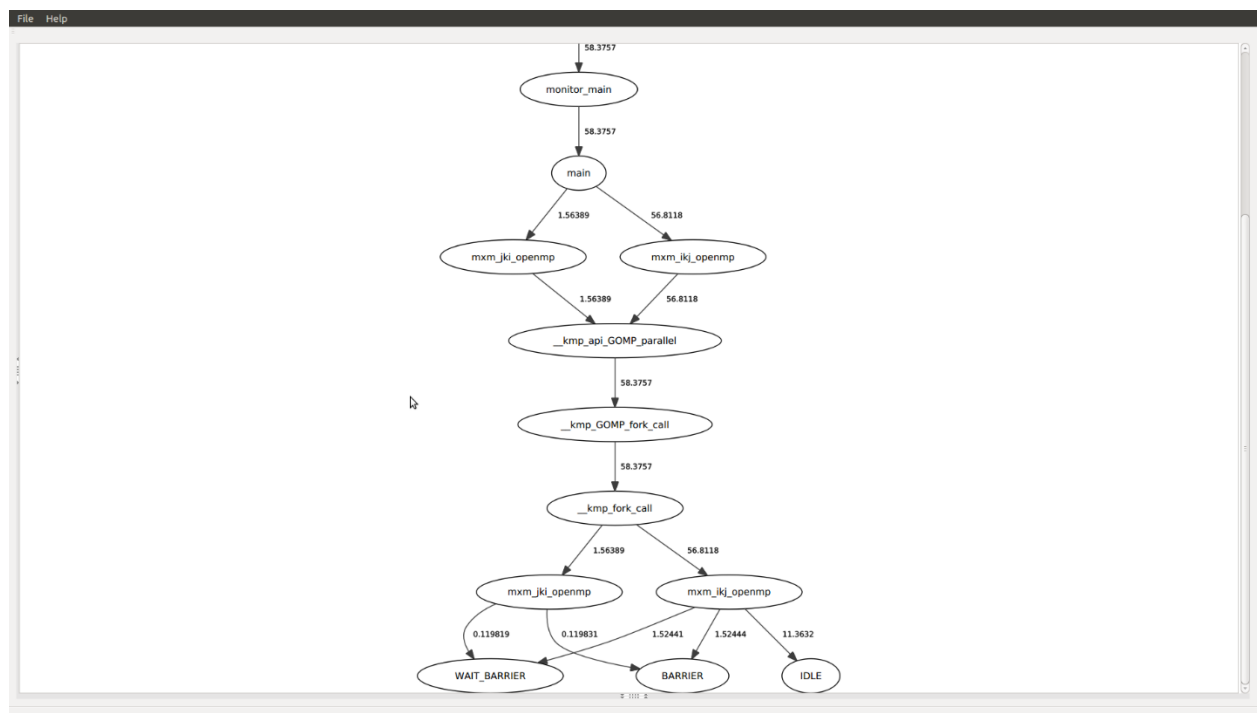


Figure 36 - calltree graph showing caller-callee relationships

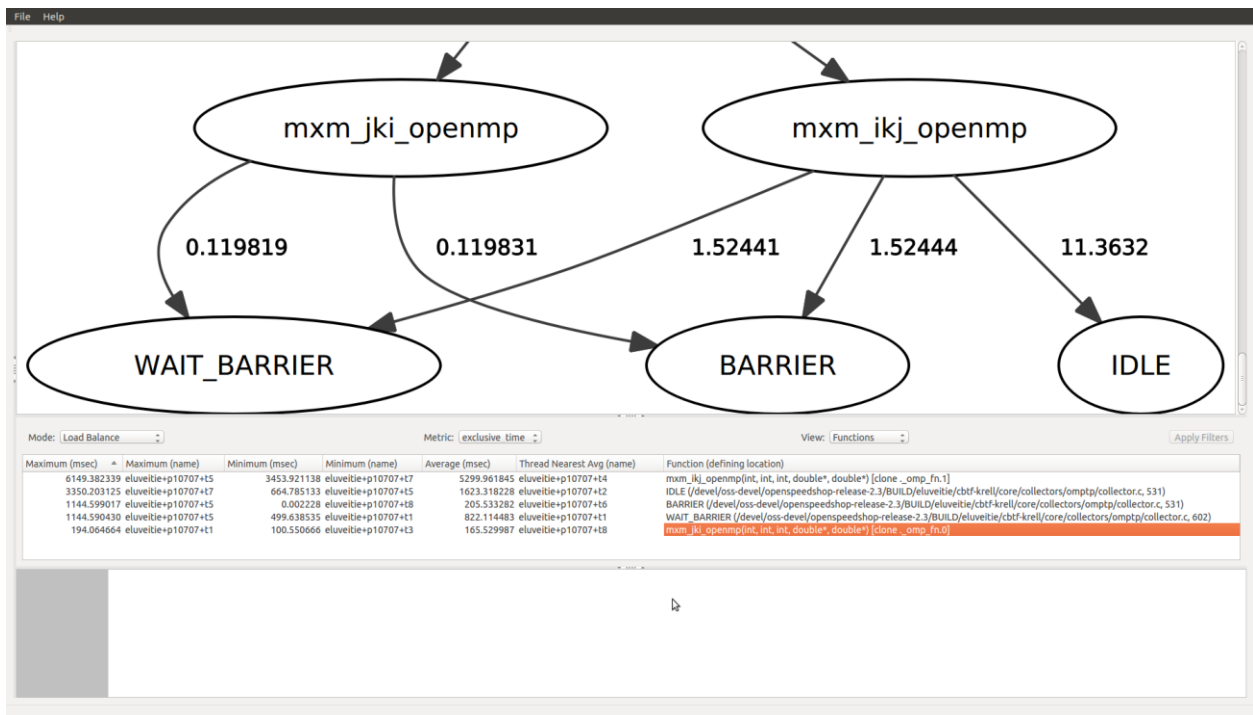


Figure 37 - omptp load balance view

13.6.1.2.7 Using the O|SS GUI to Analyze “mem” Experiment Results

Upon loading the “mem” experiment the default view appears showing a line graph of the new high-water marks along the experiment timeline. There is also a table view shown in the Metric Table View (ref Figure 38, “mem experiment default view”) showing detailed memory event information. Figure 39, “source-code with 10 memory leaks” shows the source-code in which all memory allocations are leaked and Figure 40, “source-code with 5 memory leaks” shows the source-code in which half the memory allocations are leaked.

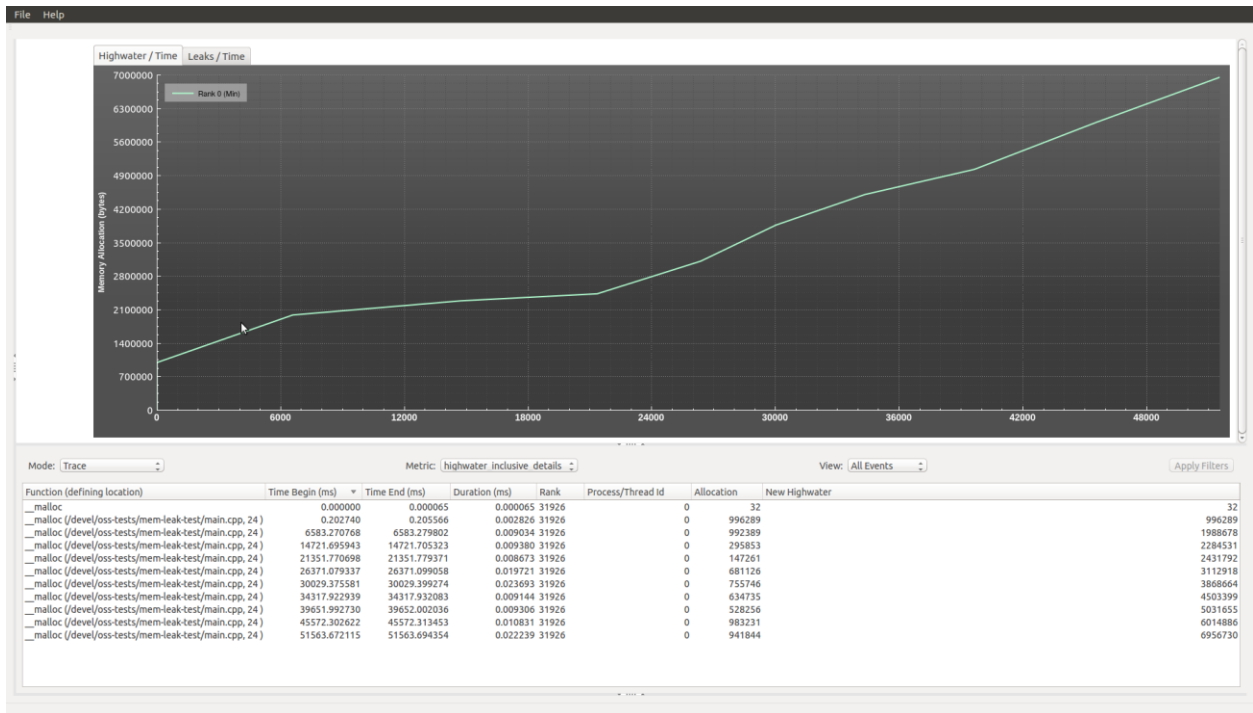


Figure 38 - mem experiment default view

```
for (int i=0; i<10; ++i) {
    char* buffer = (char *) malloc( dis(mt) );

    usleep( sleepdis(mt)*1000*1000 );
}
```

Figure 39 -

source-code with 10 memory leaks

```
for (int i=0; i<10; ++i) {
    const int allocation( dis(mt) );
    char* buffer = (char *) malloc( allocation );

    usleep( sleepdis(mt)*1000*1000 );
    if ( i % 2 == 0 ) {
        free( buffer );
    }
}
```

Figure 40 -

source-code with 5 memory leaks

Observe the Metric Table View in Figure 41, “ten memory leaks associated with Figure 39 source-code” which shows the trace metric view listing all leaked memory allocations. There are a total of 10 memory leaks. Upon the selection of one of the cells under the “Functions (defining location)” column the associated source-code (if available) will be shown in the Source-Code View underneath the Metric Table View. Figure 41 shows that line 24 of the main.cpp file was selected which caused the main.cpp file to be loaded into the Source-Code View and the view centered at line 24.

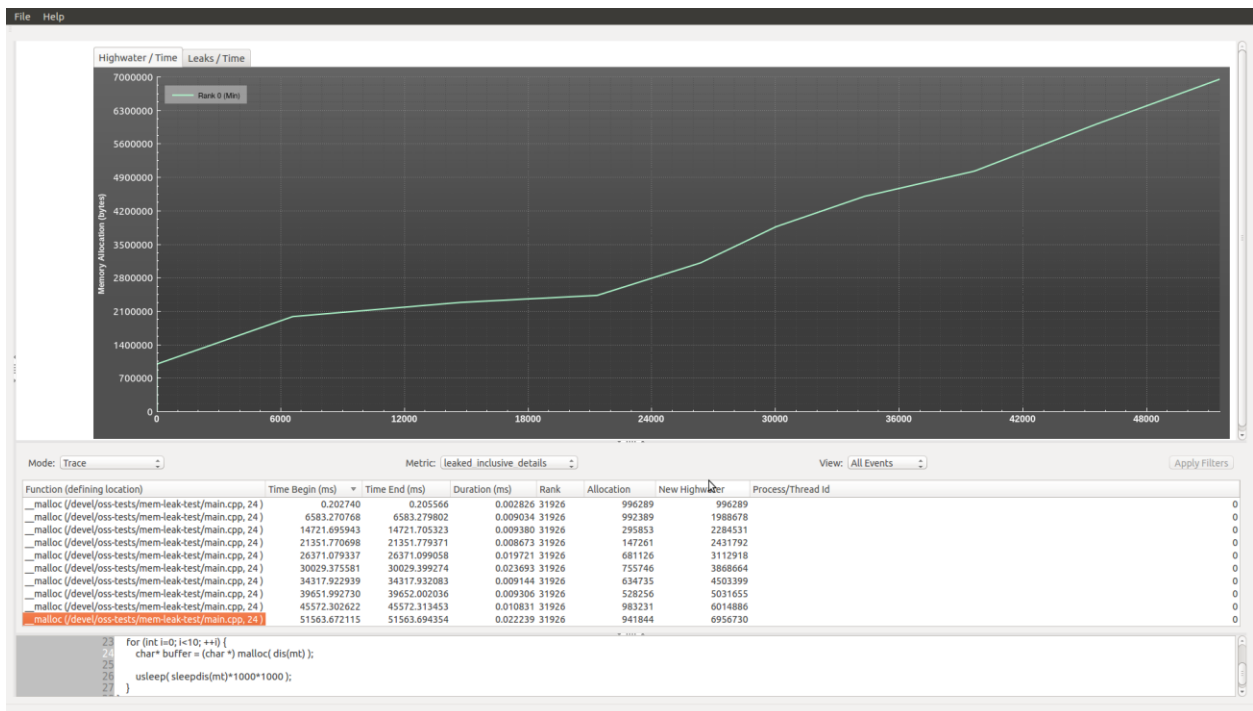


Figure 41 – ten memory leaks associated with Figure 39 source-code

Observe the Metric Table View in Figure 41, “five memory leaks associated with Figure 40 source-code” which shows the trace metric view listing a total of five leaked memory allocations. Upon the selection of one of the cells under the “Functions (defining location)” column the associated source-code (if available) will be shown in the Source-Code View underneath the Metric Table View. Figure 42 shows that line 27 of the main.cpp file was selected which caused the main.cpp file to be loaded into the Source-Code View and the view centered at line 27.

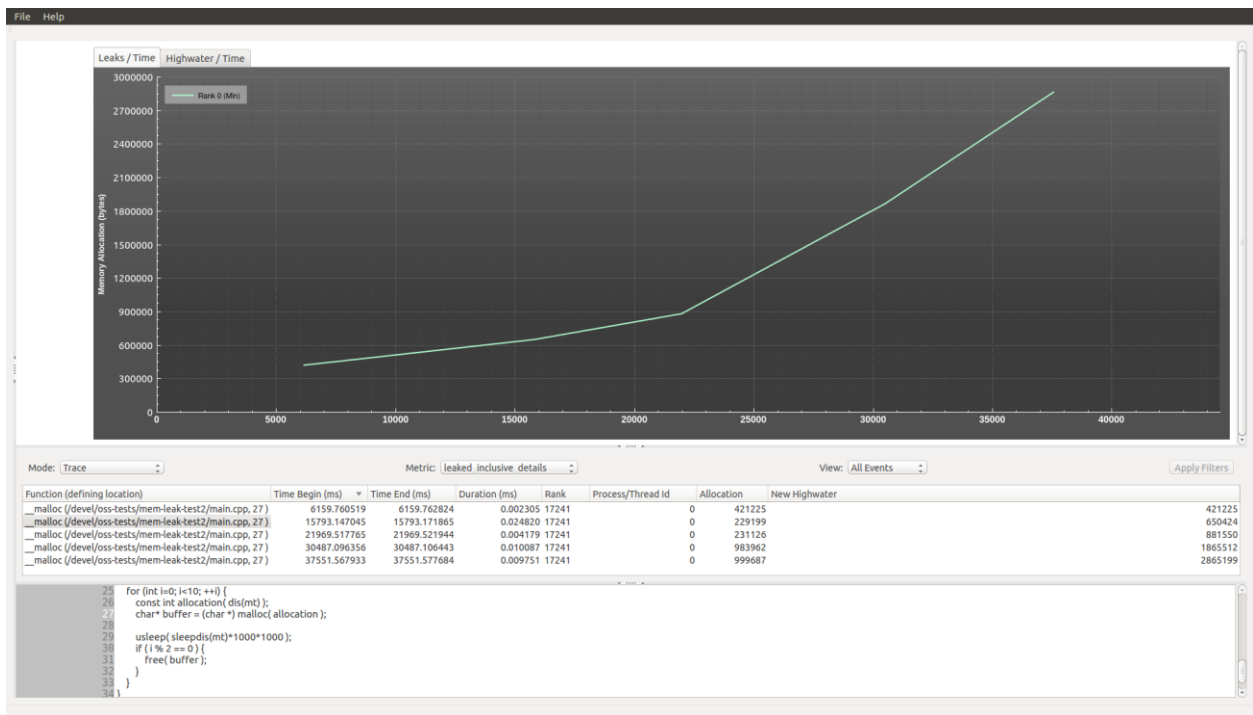


Figure 42 – five memory leaks associated with Figure 40 source-code

13.6.1.2.8 Using the O|SS GUI to Analyze “io” Experiment Results

Upon loading the “io” experiment the default view appears showing the exclusive time metric values for the functions view. Currently there is no graph generated in the Metric Plot View. However, a calltree graph showing all the caller-callee relationships captured during the experiment execution can be generated and displayed in the Metric Plot View by selecting the “CallTree” option in the “Mode” combo-box (ref Figure 43, “time metric view (with calltree graph)”).

For the calculation of metric values, the following I/O events (functions in the GNU C Library “libc”) are monitored by default: close, creat, creat64, dup, dup2, lseek, lseek64, open, open64, pipe, pread, pread64, pwrite, pwrite64, read, readv, write, writev. A subset of these can be specified when the “ossio” convenience script is executed.

Additional views of possible interest to help point out imbalance of processing between processes or ranks are the Compare By Process (ref Figure 44, “compare by process view (with calltree graph)”) and Compare By Rank (ref Figure 45, “compare by rank view (with calltree graph)”) views. These are generated by selecting the “Compare By Process” or “Compare By Rank” option in the “Mode” combo-box. In addition, the Load Balance view allows the user to see the minimum, maximum and average times for each function captured during the experiment execution. Along with the minimum and maximum time values the associated component name is

identified. For the average time value the nearest component is identified by name (ref Figure 46, “load balance view (with calltree graph)”).

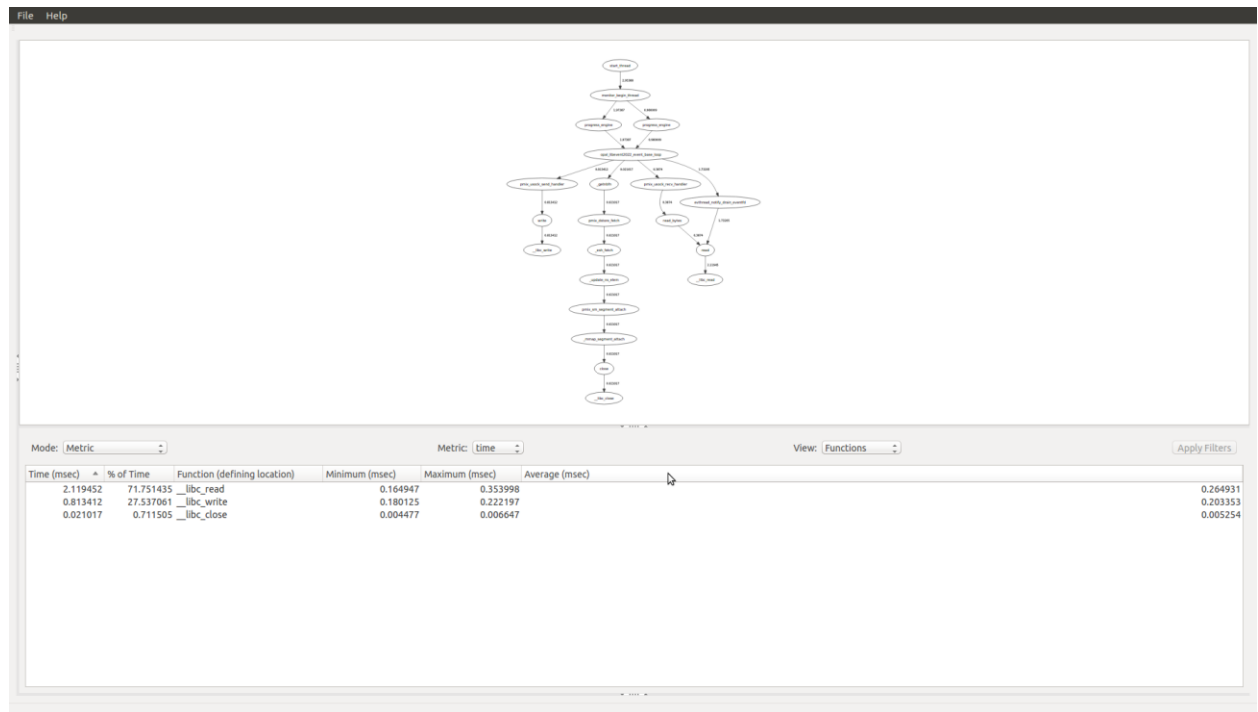


Figure 43 - time metric view (with calltree graph)

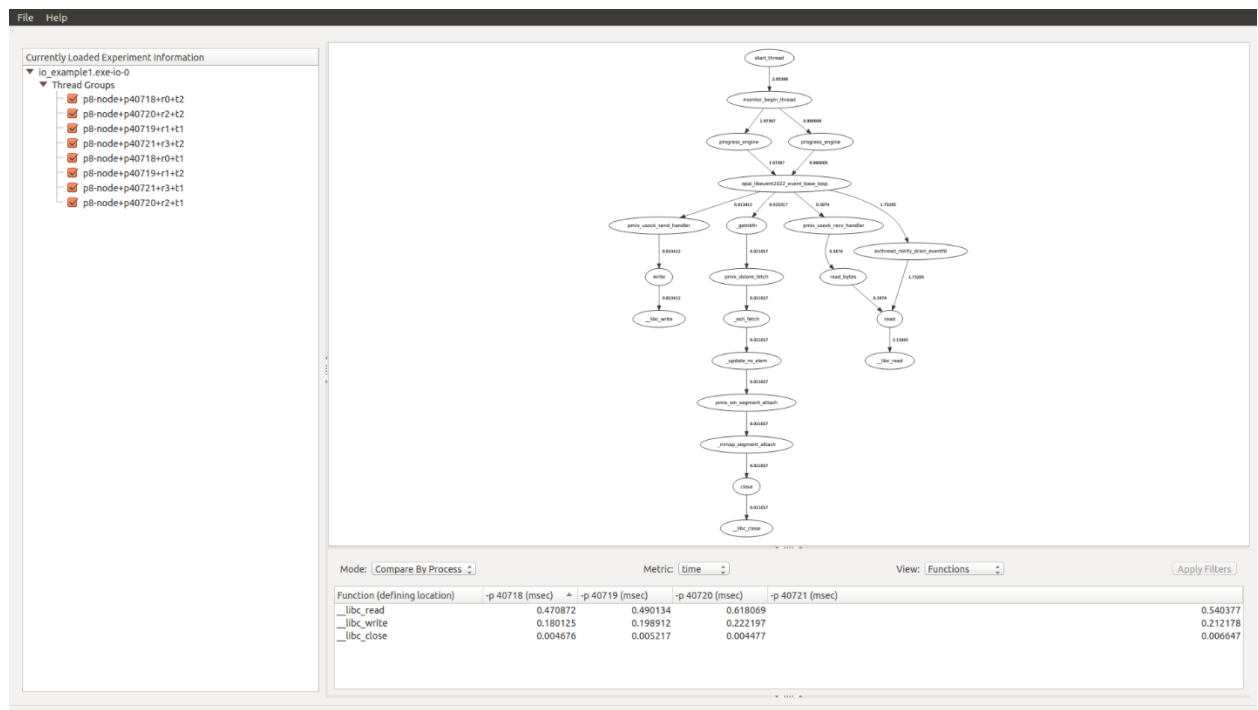


Figure 44 - compare by process view (with calltree graph)

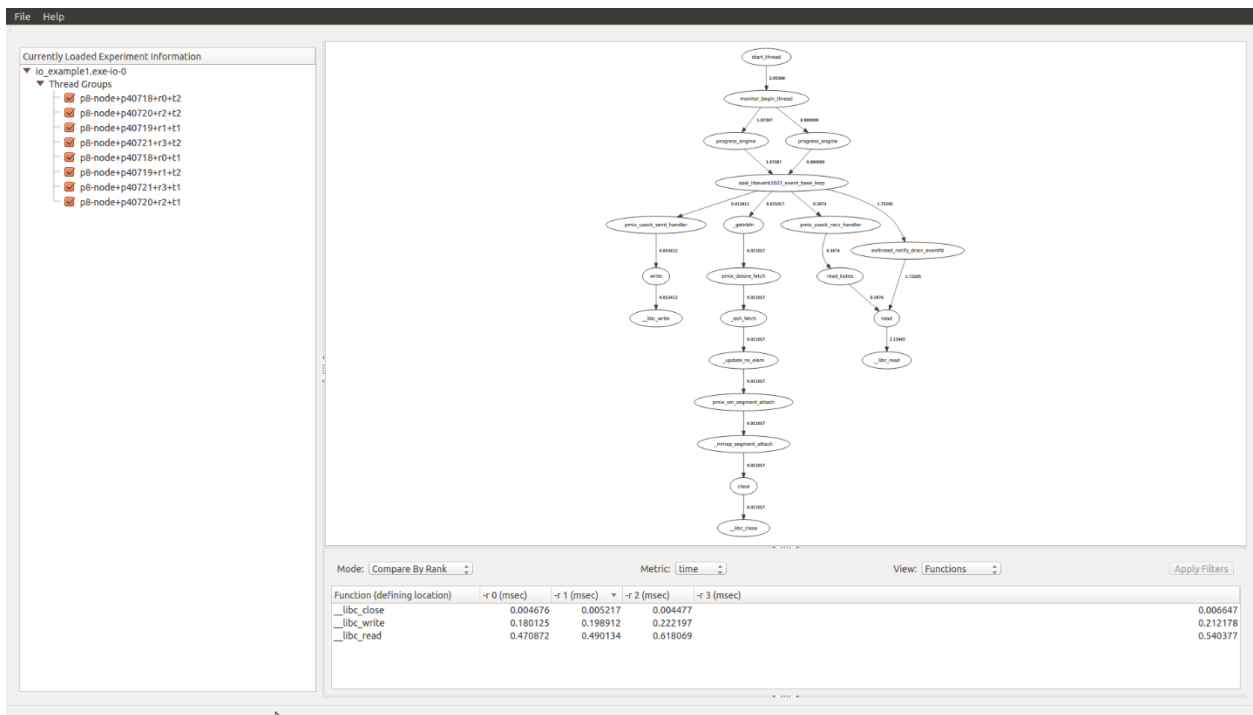


Figure 45 - compare by rank view (with calltree graph)

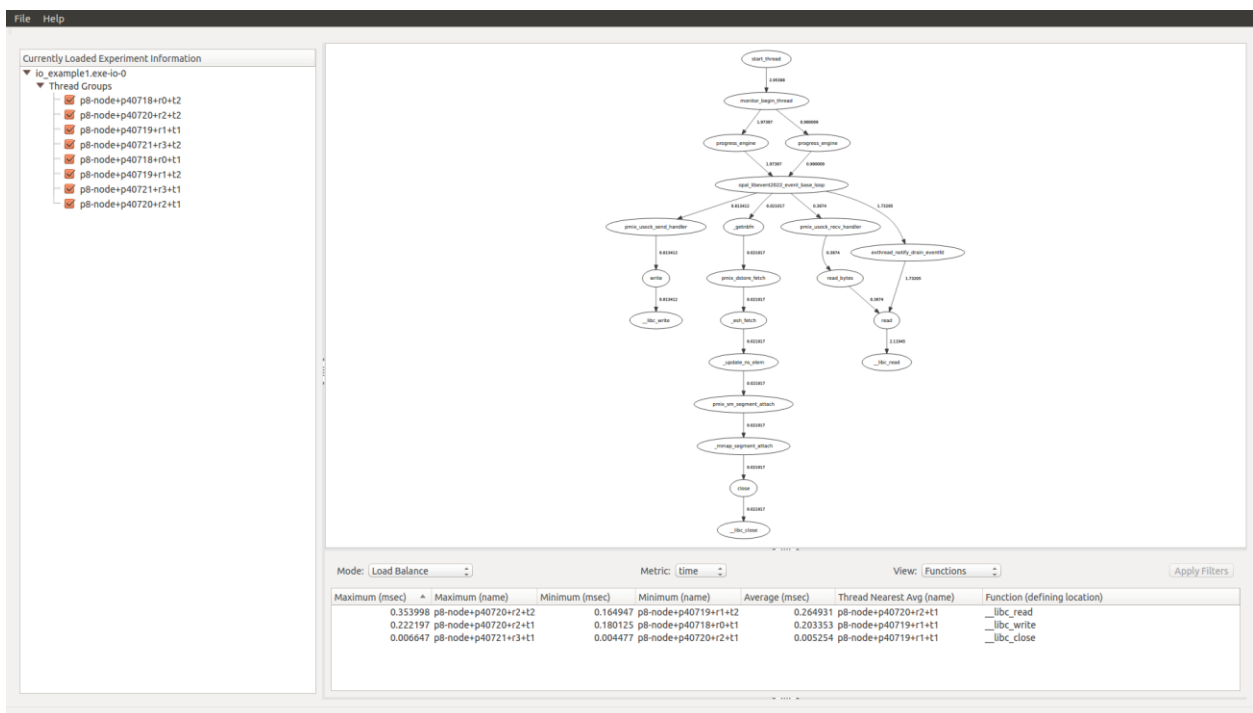


Figure 46 - load balance view (with calltree graph)

13.6.1.2.9 Using the O|SS GUI to Analyze “iop” Experiment Results

Upon loading the “iop” experiment the default view appears showing the exclusive time metric values for the functions view. Currently there is no graph generated in the Metric Plot View. However, a calltree graph showing all the caller-callee relationships captured during the experiment execution can be generated and displayed in the Metric Plot View by selecting the “CallTree” option in the “Mode” combo-box (ref Figure 47, “iop experiment time metric view (with calltree graph)”).

For the calculation of metric values, the following I/O events (functions in the GNU C Library “libc”) are monitored by default: close, creat, creat64, dup, dup2, lseek, lseek64, open, open64, pipe, pread, pread64, pwrite, pwrite64, read, readv, write, writev. A subset of these can be specified when the “ossiop” convenience script is executed.

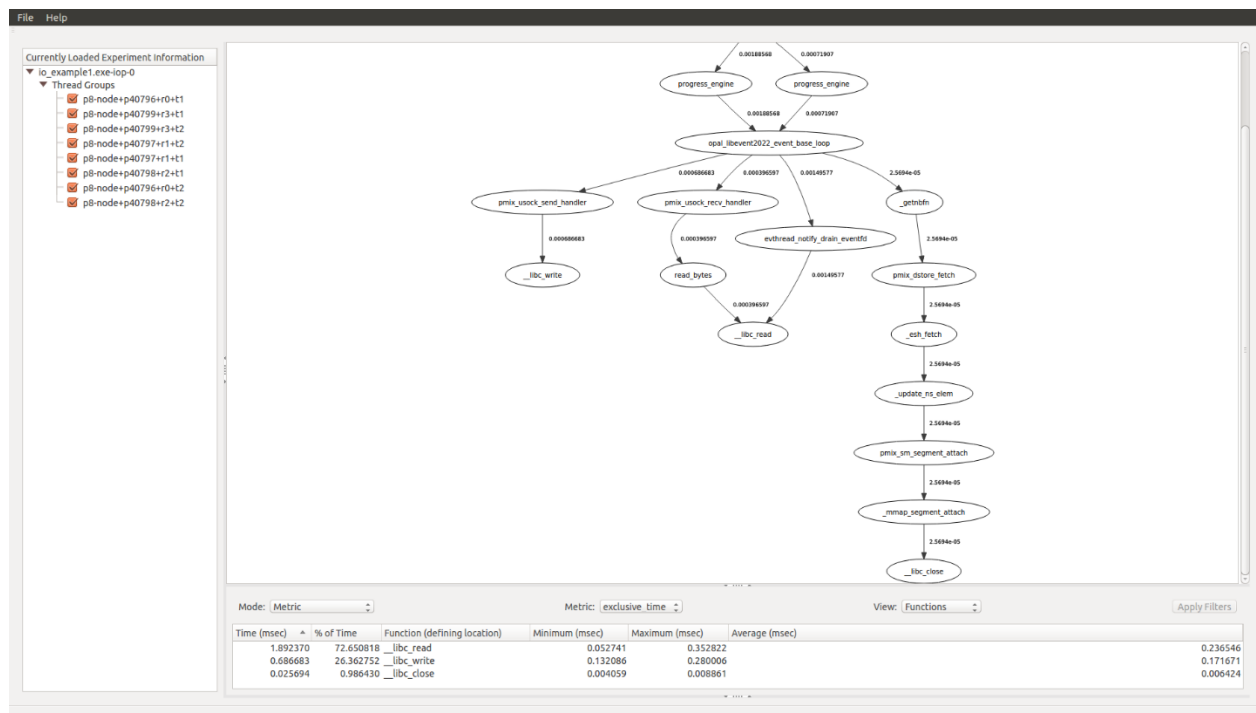


Figure 47 - iop experiment time metric view (with calltree graph)

Additional views of possible interest to help point out imbalance of processing between processes or ranks are the Compare By Process (ref Figure 48, “iop experiment compare by process view (with calltree graph)”) and Compare By Rank (ref Figure 49, “iop experiment compare by rank view (with calltree graph)”) views. These are generated by selecting the “Compare By Process” or “Compare By Rank” option in the “Mode” combo-box. In addition, the Load Balance view allows the user to see the minimum, maximum and average times for each function captured during the experiment execution. Along with the minimum and maximum time values the associated component name is identified. For the average time value the nearest

component is identified by name (ref Figure 50, “iop experiment load balance view (with calltree graph)”).

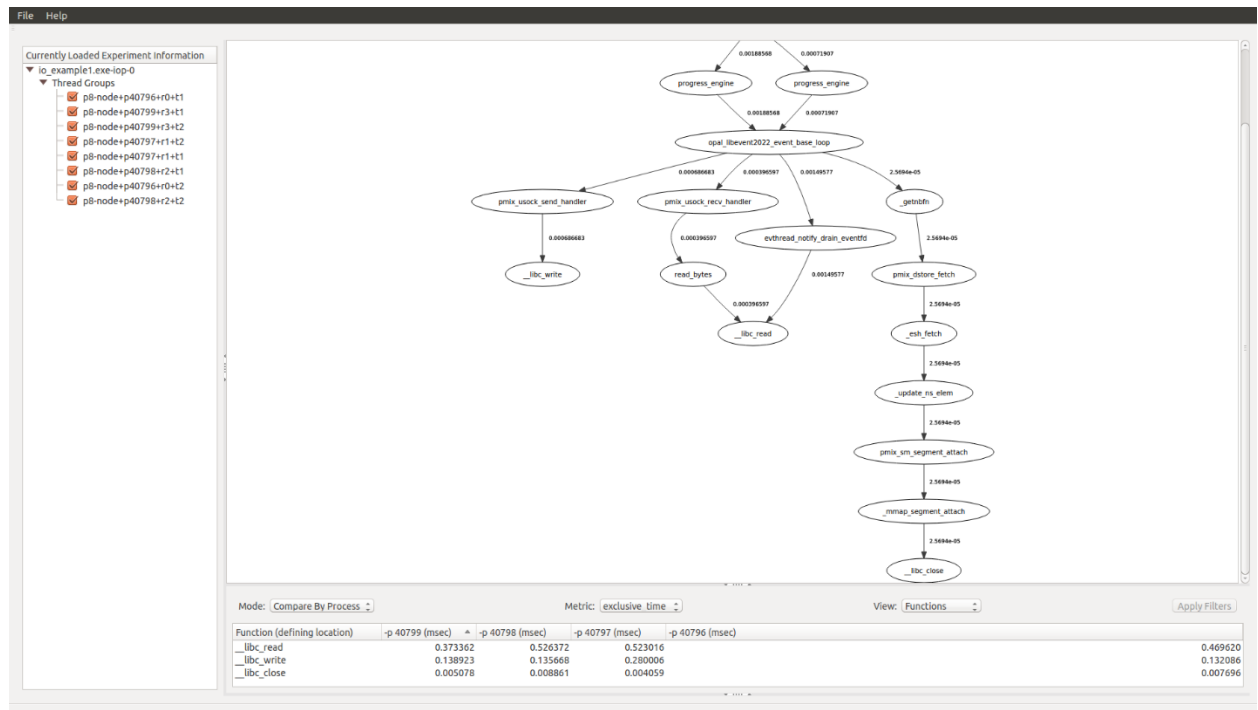


Figure 420 - iop experiment compare by process view (with calltree graph)

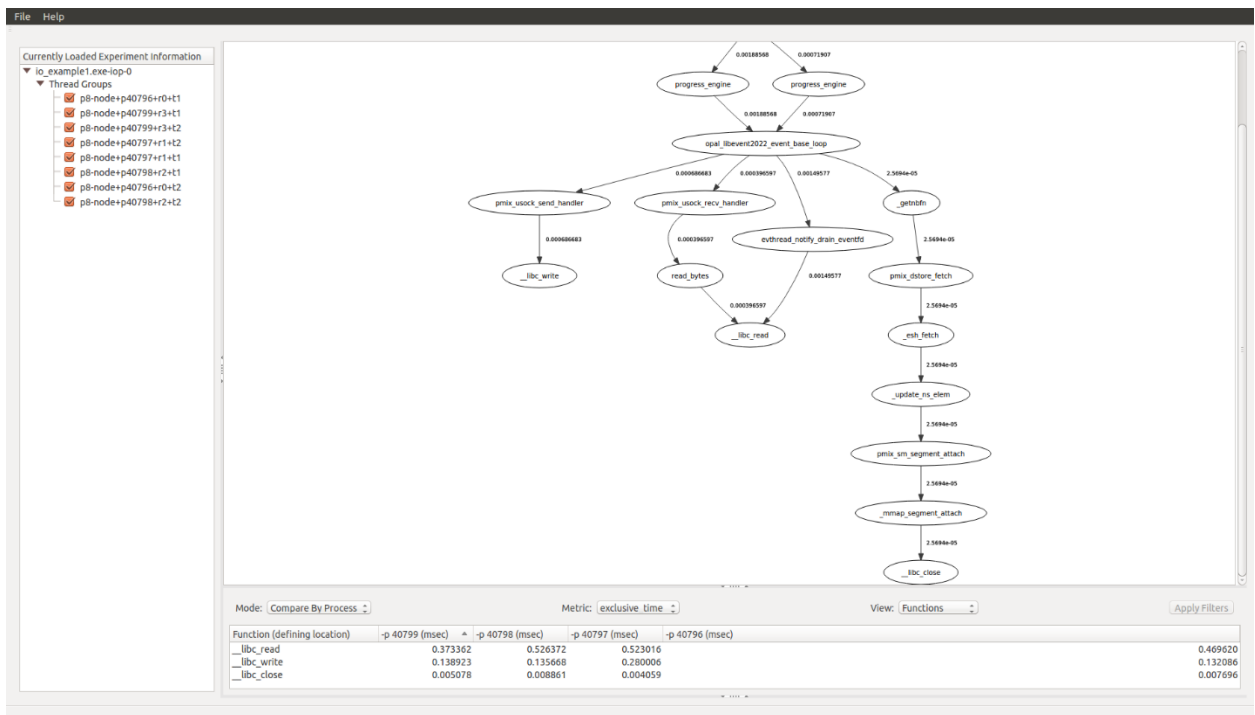


Figure 49 – iop experiment compare by rank view (with calltree graph)

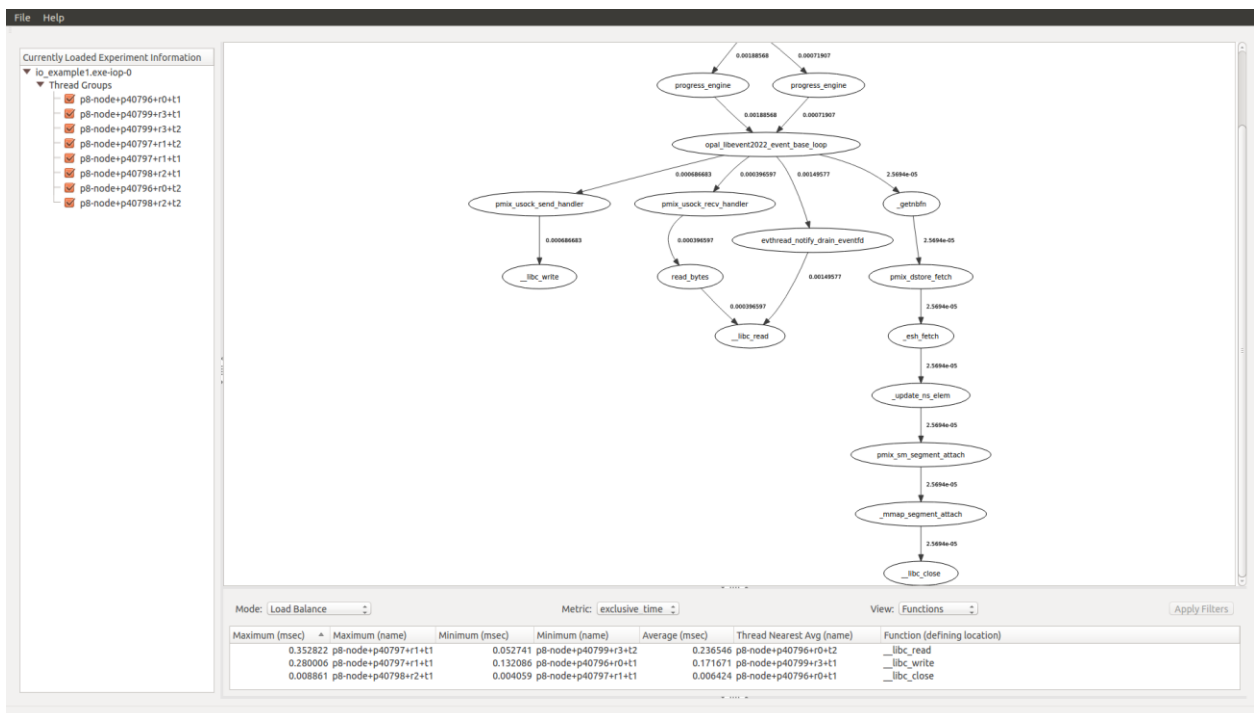


Figure 50 – iop experiment load balance view (with calltree graph)

13.6.1.2.10 Using the O|SS GUI to Analyze “iot” Experiment Results

The “iot” experiment provides extended I/O tracing capability that the “io” and “iop” experiments do not. The “iot” experiment collects additional information regarding a traced function call, the function parameters and the return value. For many of the traced I/O functions the return value is the number of bytes read or written. Since the I/O trace includes the time of the call and duration, the exact order of events can be ascertained.

Upon loading the “iot” experiment, the default view appears showing the I/O event timeline and exclusive time metric values for the functions view. The I/O event timeline appears in the Metric Plot View and maps each I/O event along a timeline covering the entire time duration of the performance data collected during the experiment execution.

The following I/O events (functions in the GNU C Library “libc”) are monitored by default: close, creat, creat64, dup, dup2, lseek, lseek64, open, open64, pipe, pread, pread64, pwrite, pwrite64, read, readv, write, writev. A subset of these can be specified when the “ossiot” convenience script is executed.

Additional views of interest to help point out imbalance of processing between processes or ranks are the Compare By Process (ref Figure 52, “iot experiment compare by process view”) and Compare By Rank (ref Figure 53, “iot experiment compare by rank view”) views. These are generated by selecting the “Compare By Process” or “Compare By Rank” option in the “Mode” combo-box. In addition, the Load Balance view allows the user to see the minimum, maximum and average times for each function captured during the experiment execution. Along with the minimum and maximum time values the associated component name is identified. For the average time value the nearest component is identified by name (ref Figure 54, “iot experiment load balance view”).

The calltree graph showing all the caller-callee relationships captured during the experiment execution can be generated and displayed in the Metric Plot View by selecting the “CallTree” option in the “Mode” combo-box (ref Figure 55, “iot experiment calltree graph”).

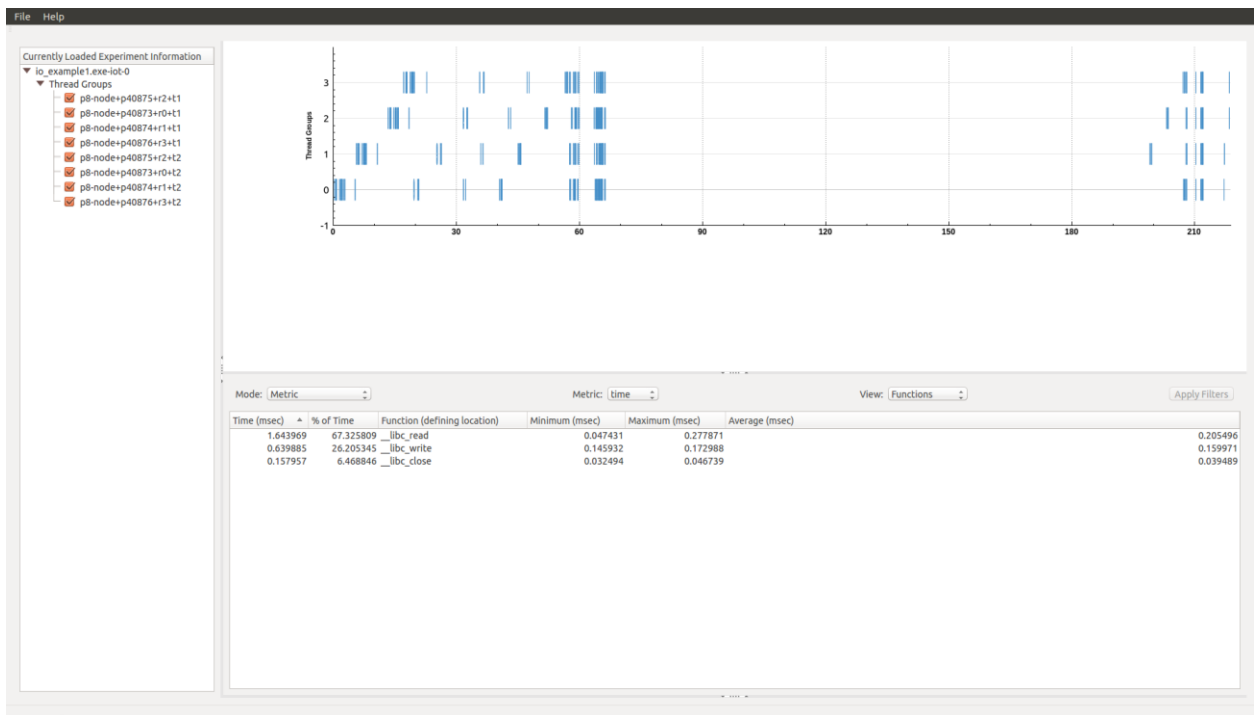


Figure 51 - iot experiment default view

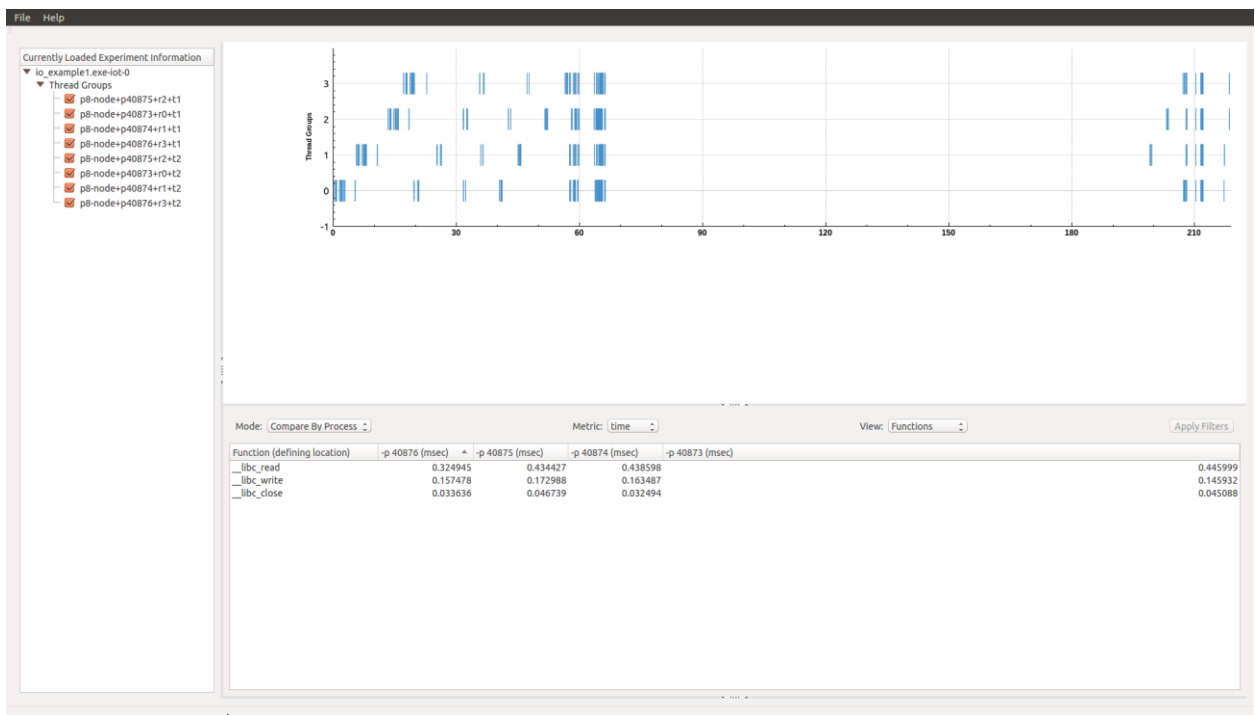


Figure 52 - iot experiment compare by process view

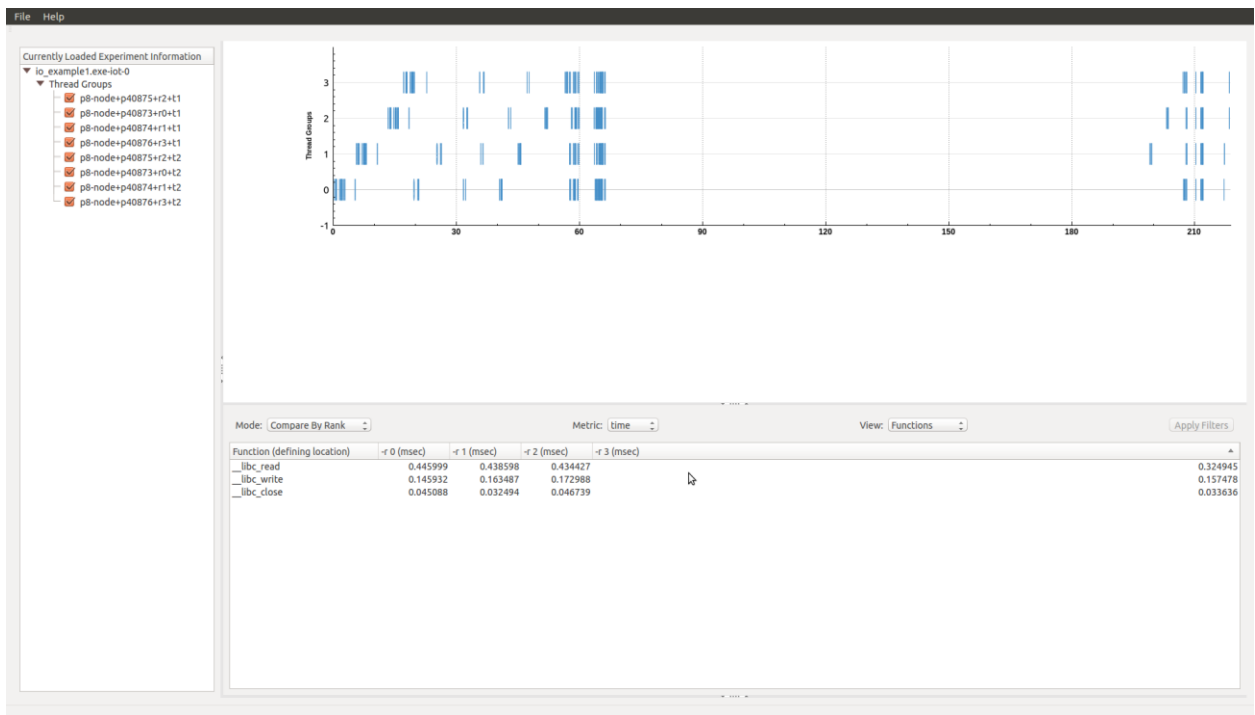


Figure 53 - iot experiment compare by rank view

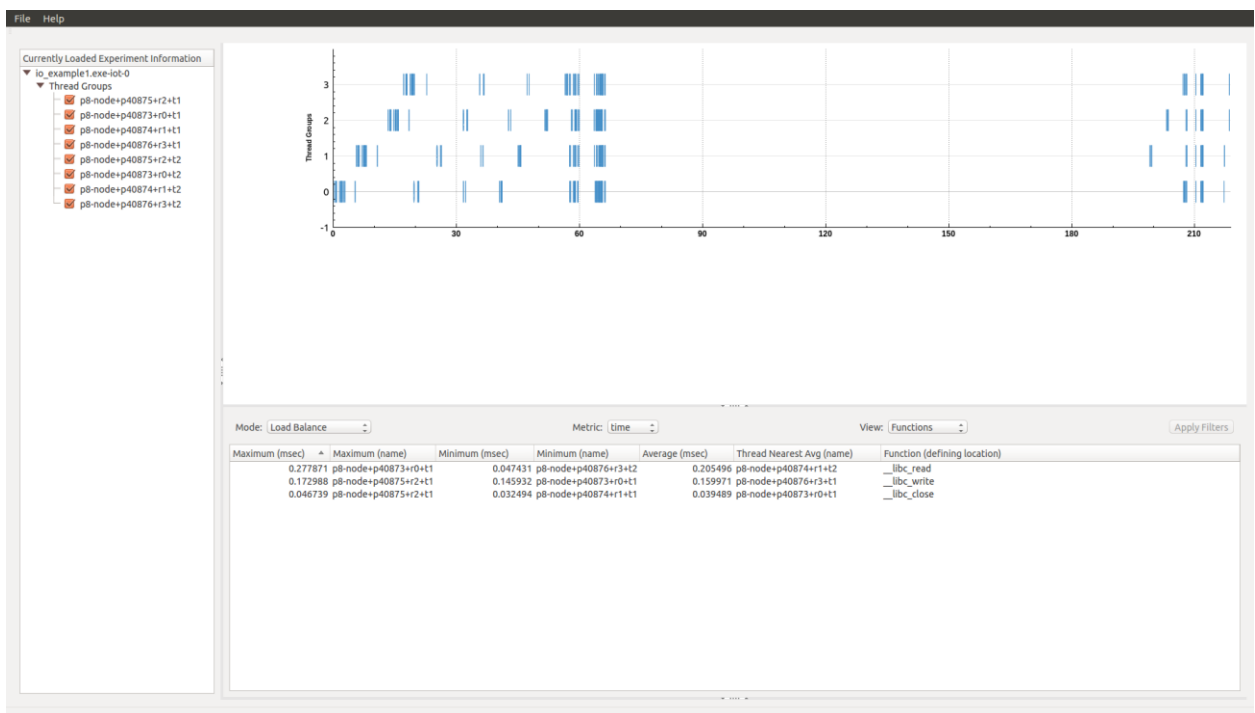


Figure 54 - iot experiment load balance view

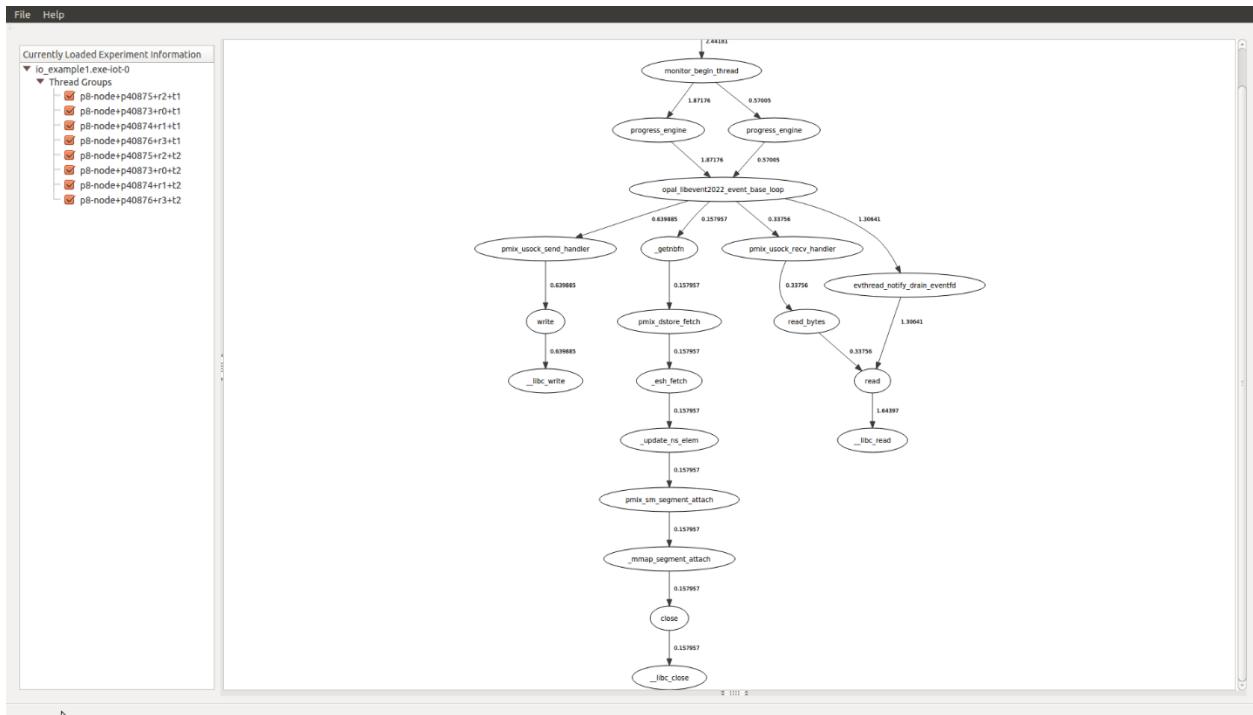


Figure 55 - iot experiment calltree graph

13.6.1.2.11 Using the O|SS GUI to Analyze “mpi” Experiment Results

Upon loading the “mpi” experiment the default view appears showing the exclusive time metric values for the functions view. Currently there is no graph generated in the Metric Plot View. However, a calltree graph showing all the caller-callee relationships captured during the experiment execution can be generated and displayed in the Metric Plot View by selecting the “CallTree” option in the “Mode” combo-box (ref Figure 56, “mpi experiment default view (with calltree graph)”).

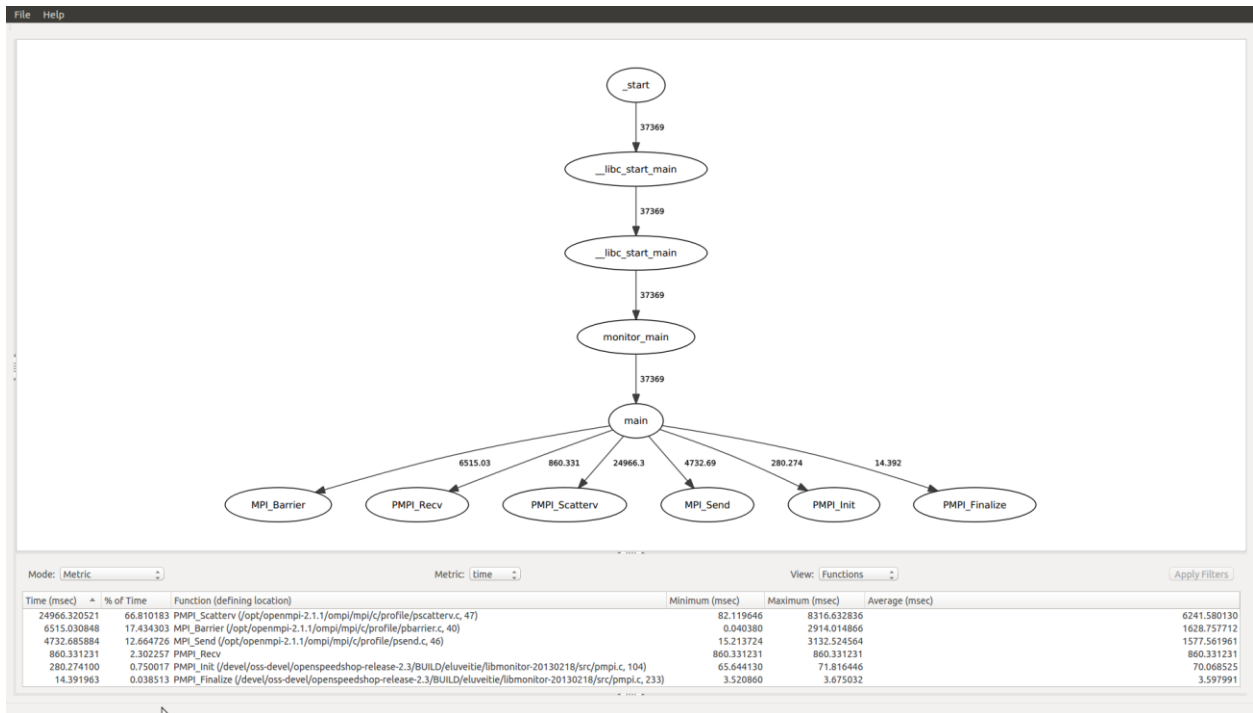


Figure 56 - mpi experiment default view (with calltree graph)

Additional views of possible interest to help point out imbalance of processing between processes or ranks are the Compare By Process (ref Figure 57, “compare by process view (with calltree graph)”; Compare By Rank (ref Figure 58, “compare by rank view (with calltree graph)”) views; and Compare (ref Figure 59, “compare view (with calltree graph)”) views. These are generated by selecting the “Compare By Process”, “Compare By Rank” or “Compare” option in the “Mode” combo-box. In addition, the Load Balance view allows the user to see the minimum, maximum and average times for each function captured during the experiment execution. Along with the minimum and maximum time values the associated component name is identified. For the average time value the nearest component is identified by name (ref Figure 60, “load balance view (with calltree graph)”).

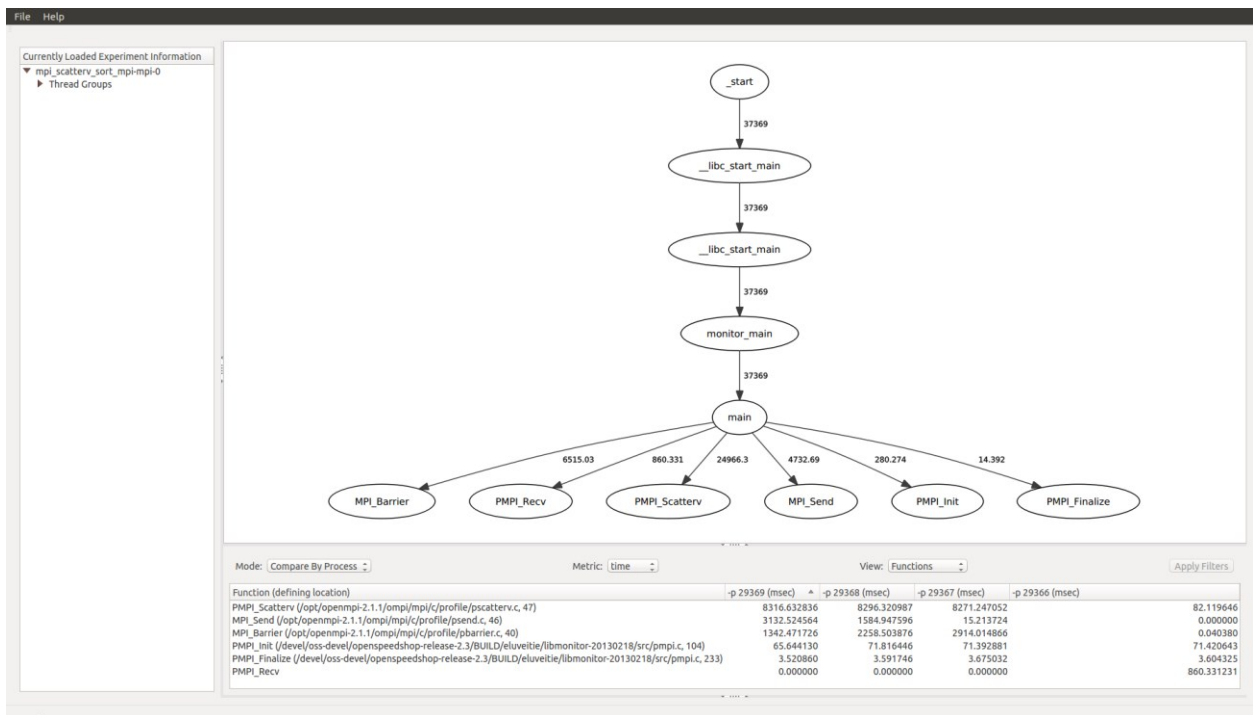


Figure 57 - compare by process view (with calltree graph)

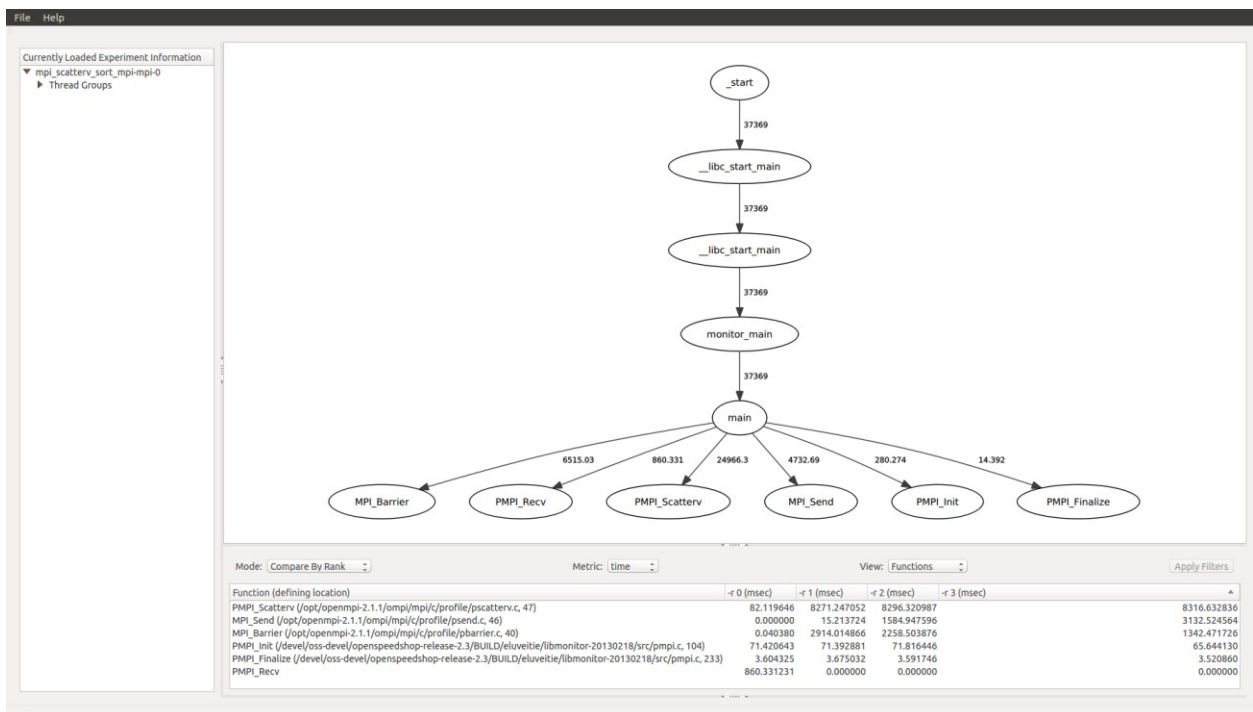


Figure 58 - compare by rank view (with calltree graph)

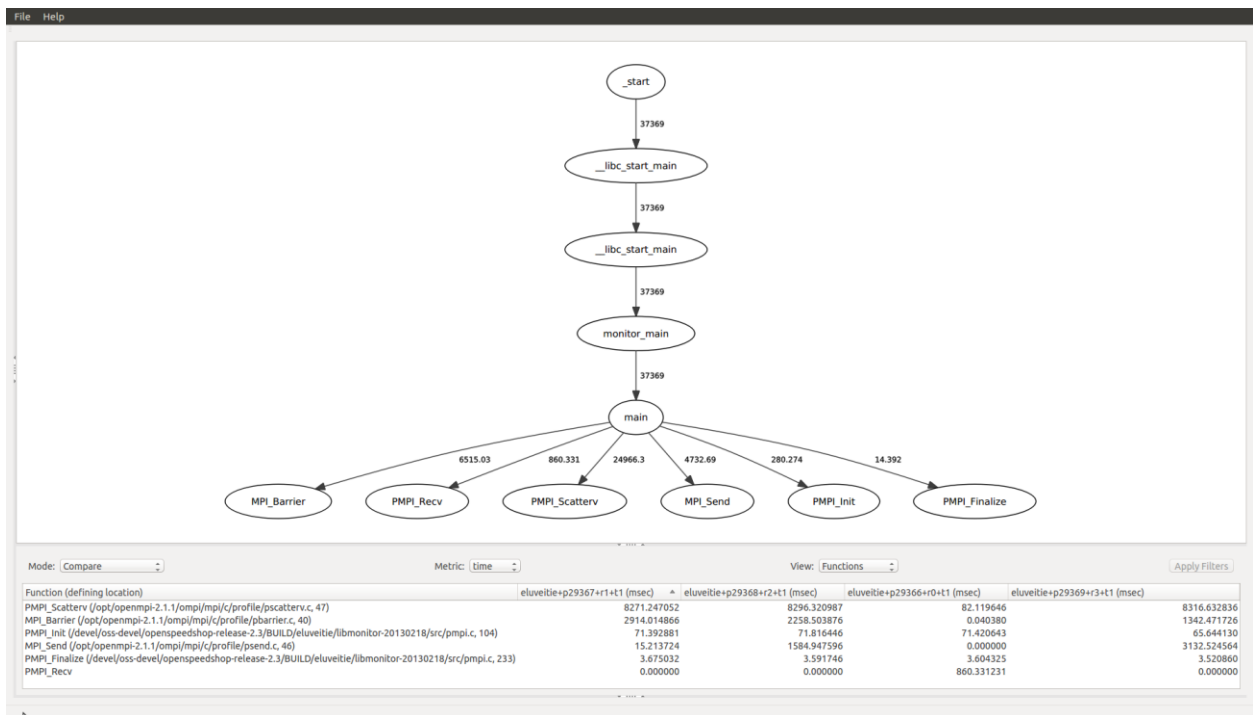


Figure 59 - compare view (with calltree graph)

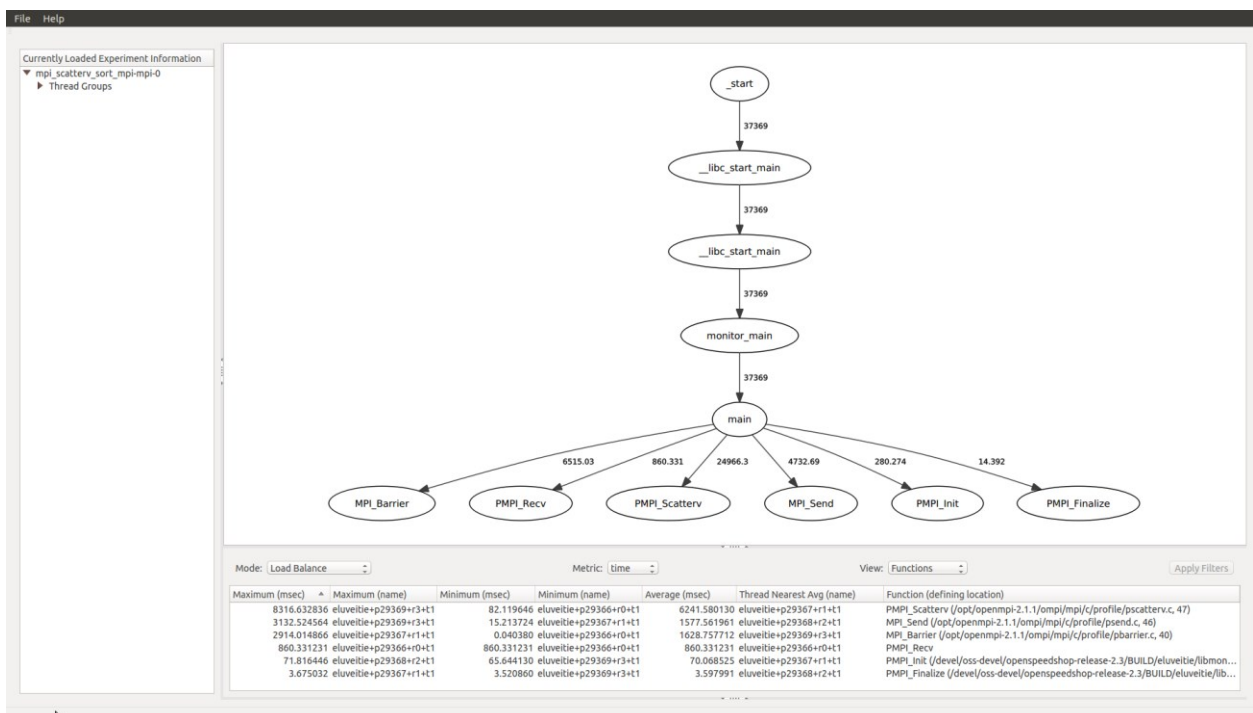


Figure 60 - load balance view (with calltree graph)

13.6.1.2.12 Using the O|SS GUI to Analyze “mpip” Experiment Results

Upon loading the “mpip” experiment the default view appears showing the exclusive time metric values for the functions view. Currently there is no graph generated in the Metric Plot View. However, a calltree graph showing all the caller-callee relationships captured during the experiment execution can be generated and displayed in the Metric Plot View by selecting the “CallTree” option in the “Mode” combo-box (ref Figure 61, “mpip experiment default view (with calltree graph)”).

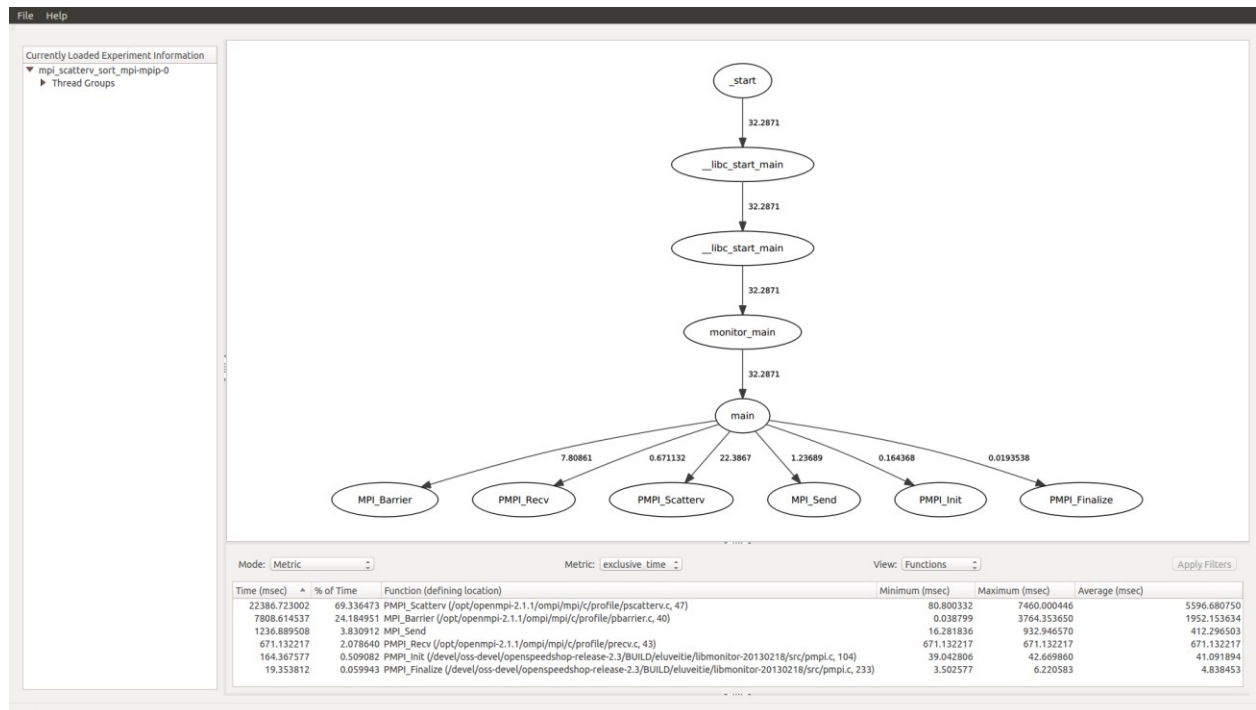


Figure 61 - mpip experiment default view (with calltree graph)

Additional views of possible interest to help point out imbalance of processing between processes or ranks are the Compare By Process (ref Figure 62, “compare by process view (with calltree graph)”); Compare By Rank (ref Figure 63, “compare by rank view (with calltree graph)”); and Compare (ref Figure 64, “compare view (with calltree graph)”); views. These are generated by selecting the “Compare By Process”, “Compare By Rank” or “Compare” option in the “Mode” combo-box. In addition, the Load Balance view allows the user to see the minimum, maximum and average times for each function captured during the experiment execution. Along with the minimum and maximum time values the associated component name is identified. For the average time value the nearest component is identified by name (ref Figure 65, “load balance view (with calltree graph)”).

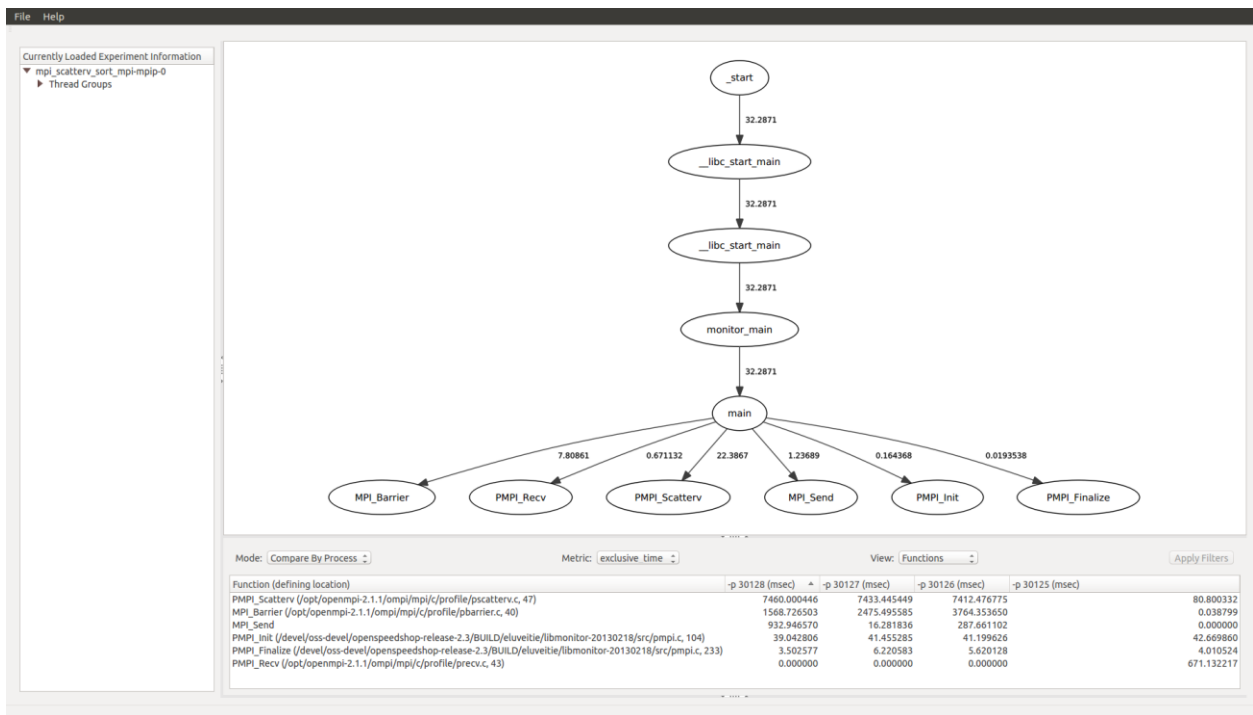


Figure 62 - compare by process view (with calltree graph)

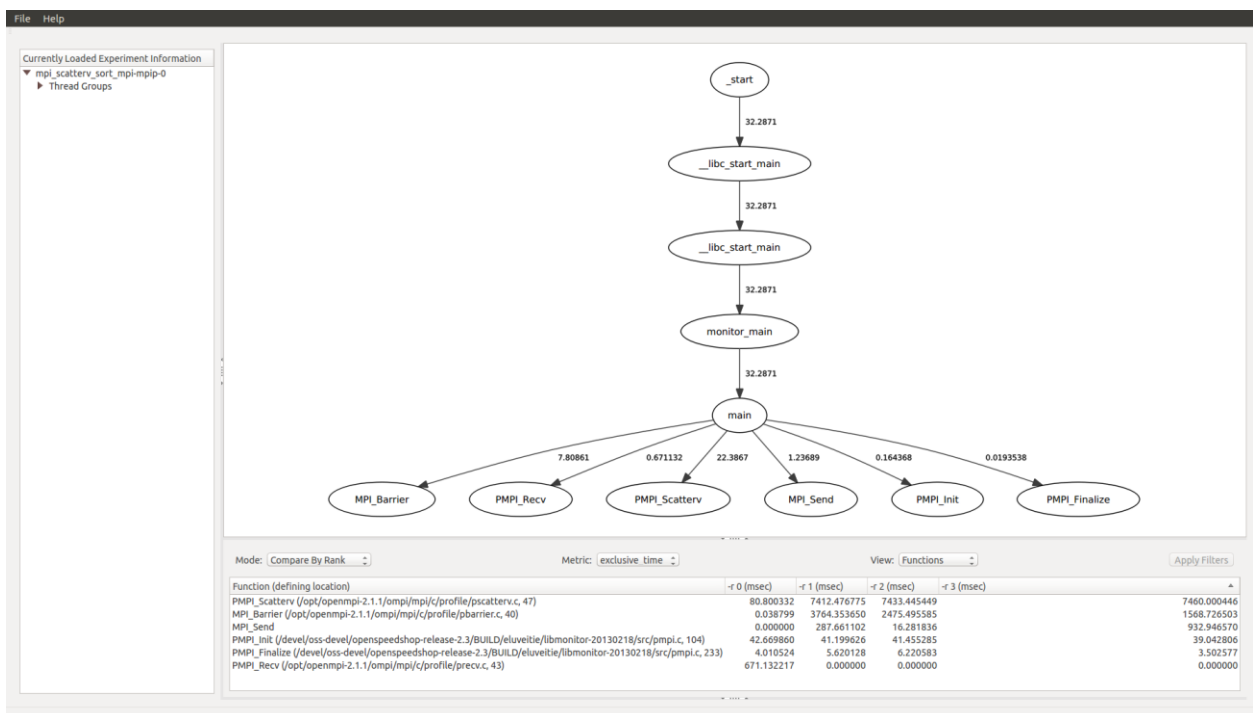


Figure 63 - compare by rank view (with calltree graph)

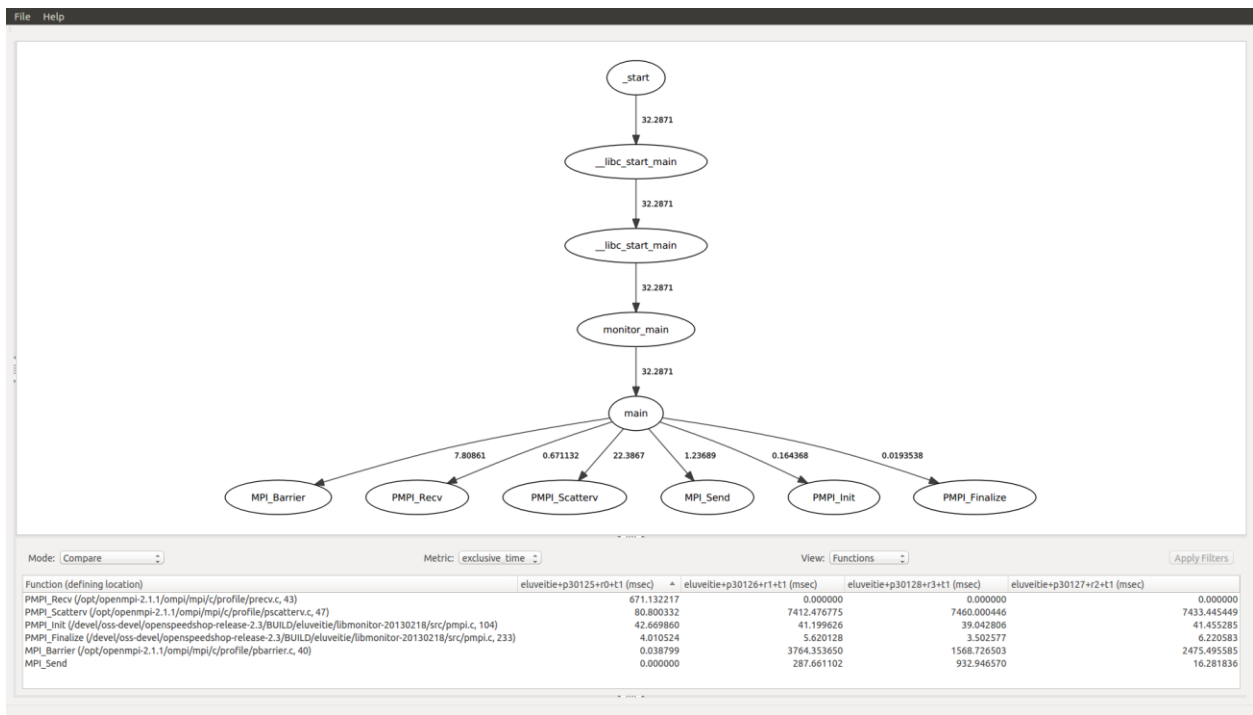


Figure 64 - compare view (with calltree graph)

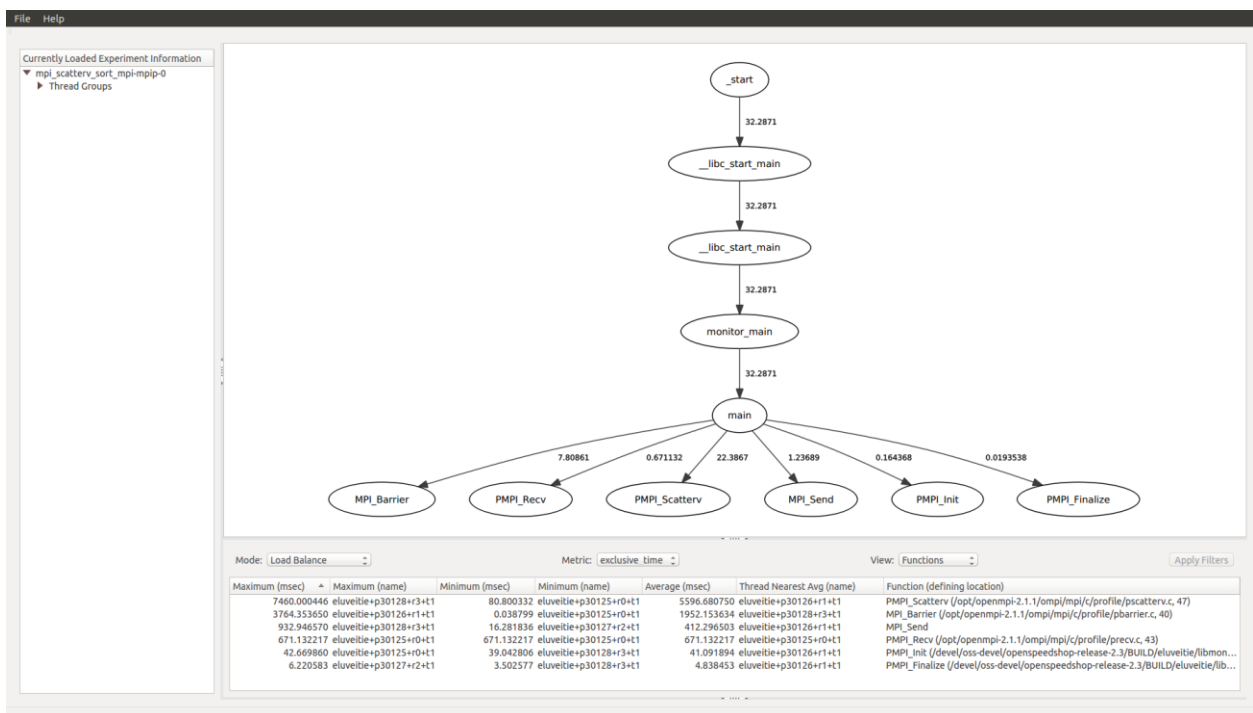


Figure 65 - load balance view (with calltree graph)

13.6.1.2.12 Using the O|SS GUI to Analyze “mpit” Experiment Results

Upon loading the “mpit” experiment the default view appears showing the MPI event timeline and exclusive time metric values for the functions view (ref Figure 66, “mpit experiment default view”).

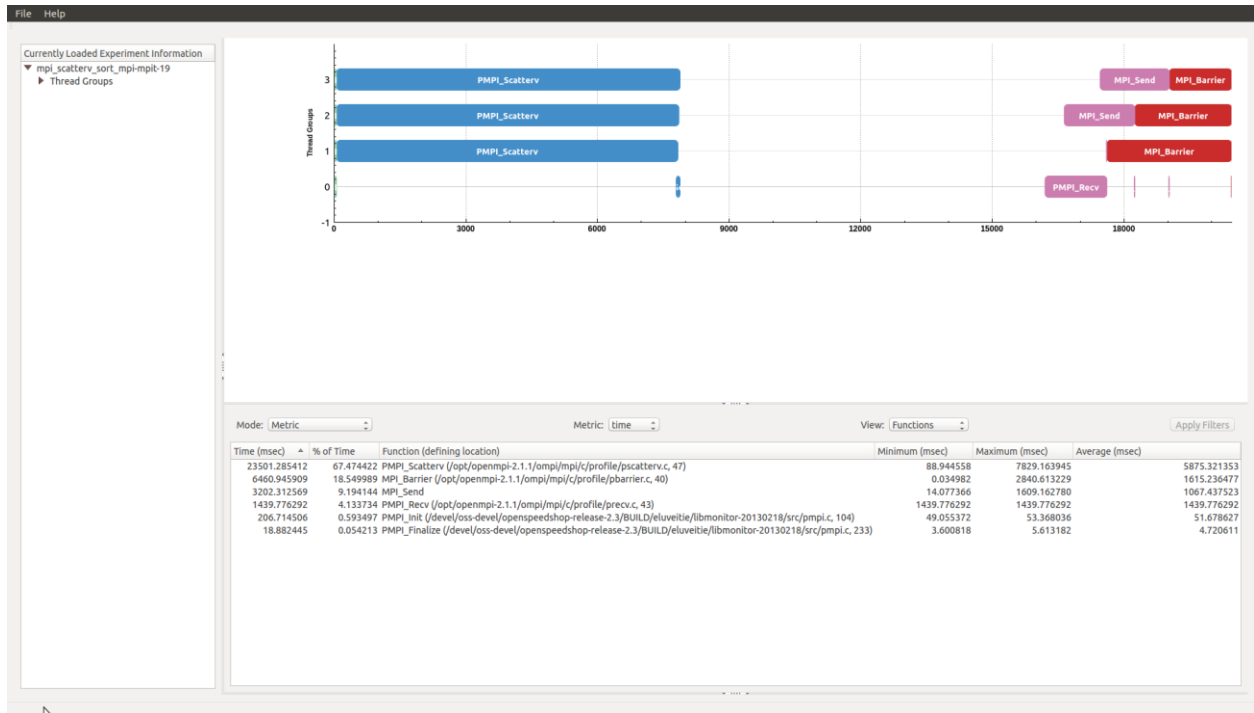


Figure 66 - mpit experiment default view

The MPI event timeline shows every MPI function call that occurred within the given graph time range. For the default view, the graph time range is the full experiment time span. Each MPI function is drawn as a rounded rectangle where the left edge is at the time the MPI function was called and the right edge is when the MPI function call completed. Thus, the length of the rectangle can provide visual cues as to the magnitude of the MPI call duration.

Select the “Trace” option from the “Mode” combo-box to show a list of detailed information regarding each MPI event in the Metric Table View - the MPI function name, the time the function was invoked and completed (in milliseconds from the relative beginning of the experiment), the duration (in milliseconds), the rank from which the MPI function was invoked, the destination rank, size of the message (in bytes) and the MPI function return value (ref Figure 67, “MPI event list in Metric Table View”).

The graph time range can be manipulated by holding the left-mouse button and scrolling the mouse wheel forward to zoom into the graph and scrolling the mouse wheel backward to zoom out. The graph range can be panned to the left or right by

holding the left-mouse button down and sliding the mouse to the left or right. As the visible time range is updated by the user, the list of MPI events in the Metric Table View is updated to match the visible time range (ref Figure 68, “MPI event list in Metric Table View (filtered to graph range – from experiment origin)”) and (ref Figure 69, “MPI event list in Metric Table View (filtered to graph range – at experiment end range)”).

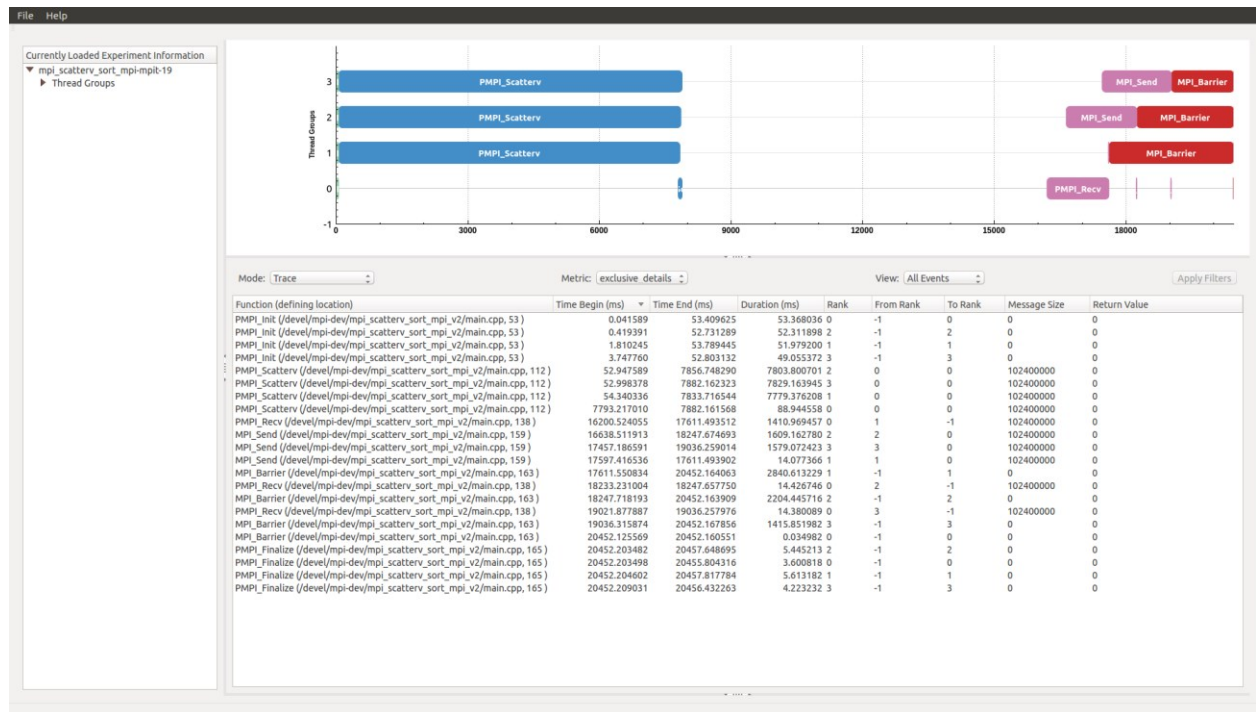


Figure 67 - MPI event list in Metric Table View

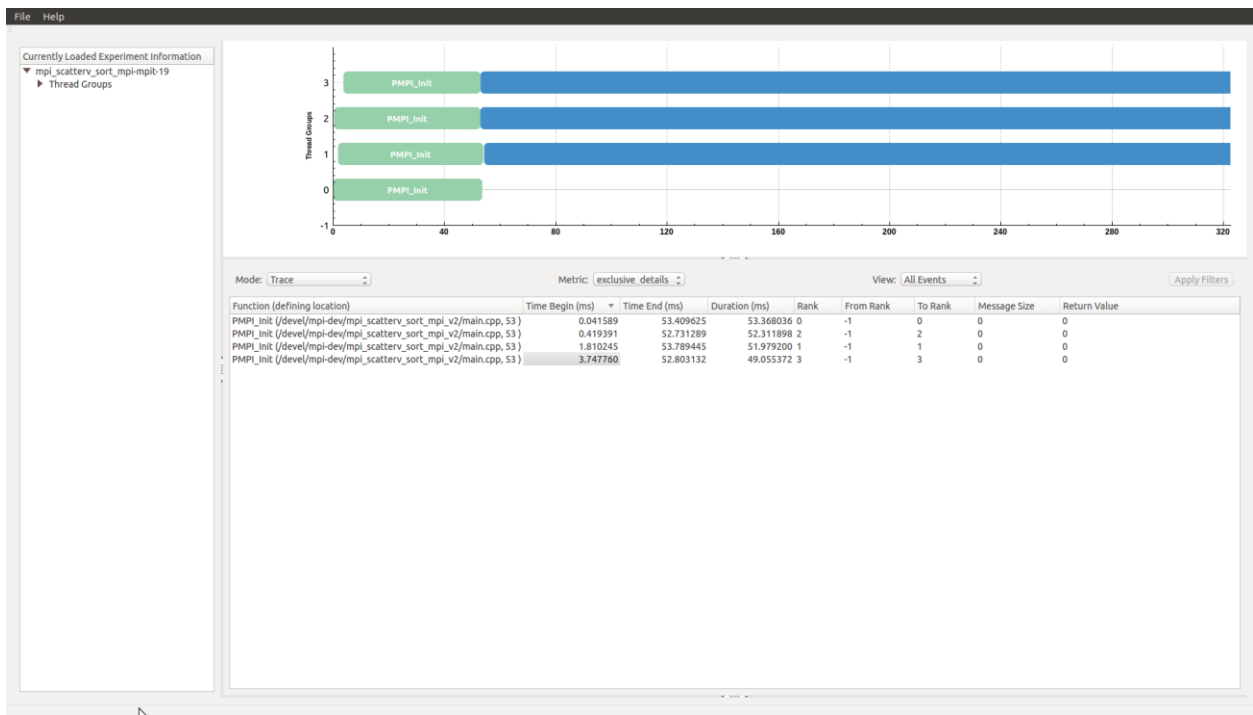


Figure 68 - MPI event list in Metric Table View (filtered to graph range – from experiment origin)

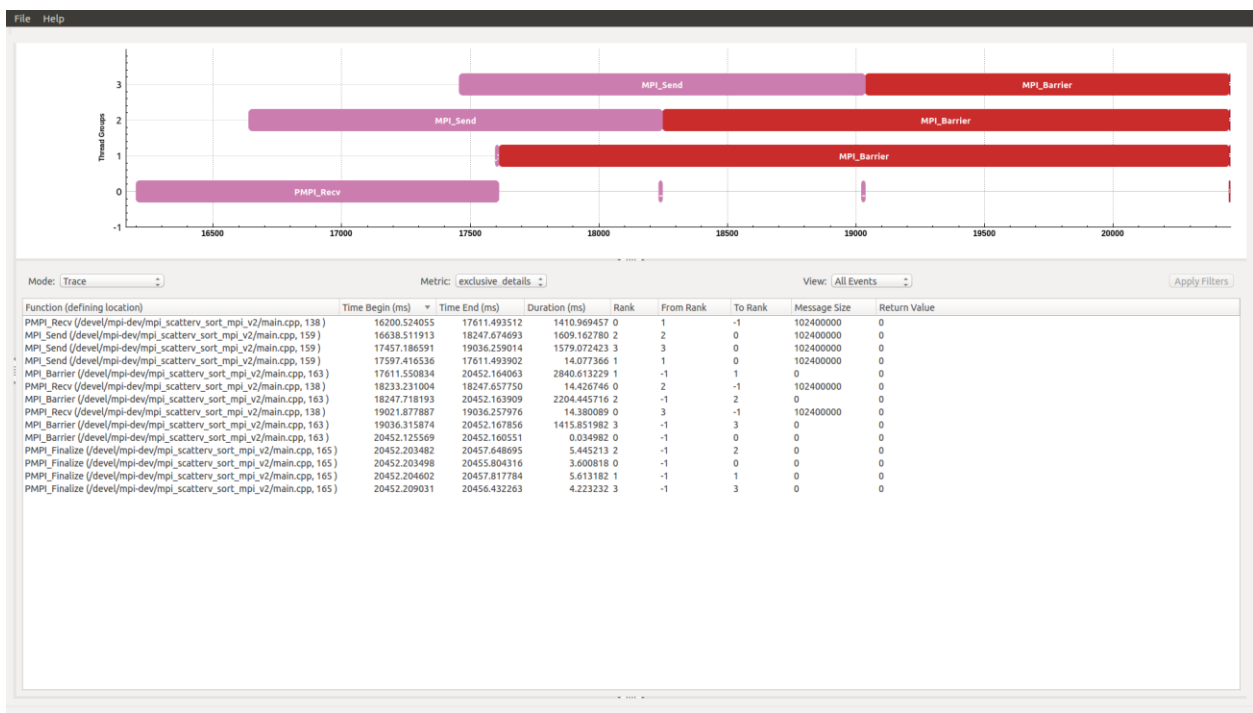


Figure 621 - MPI event list in Metric Table View (filtered to graph range – at experiment end range)

Once the MPI event list is available in the Metric Table View, if an item under the “Time Begin (ms)” or “Time End (ms)” table column is selected, the corresponding MPI event in the graph timeline is highlighted inside a slightly larger rounded yellow rectangle (ref Figure 70, “locating MPI event in graph timeline”). In addition,

as can be seen in Figure 70, a dashed bounding rectangle is also drawn to help locate the event within a crowded event timeline (ref Figure 71, “locating MPI event in a crowded MPI event timeline”).

The dashed bounding rectangle remains visible for 10 seconds during which time the graph may be zoomed into the area being highlighted (ref Figure 72 – “Using highlighting cues to zoom into selected event”). For the “Time Begin (ms)” item selected in Figure 70, once the graph has been zoomed to bring the particular MPI event into closer view, it can be seen that the event is an MPI_WaitAll call. Sometimes even after the graph has been zoomed to the fullest extent the name of the MPI function call may not be visible because the MPI function rectangle is still too small to have visible text. However, as the graph range is manipulated, in this case by zooming into the graph (i.e. reducing the visible graph range), the contents of the Metric Table View are filtered to the visible graph range so that the applicable MPI function name may be determined (ref Figure 72, “Using highlighting cues to zoom into selected event”).

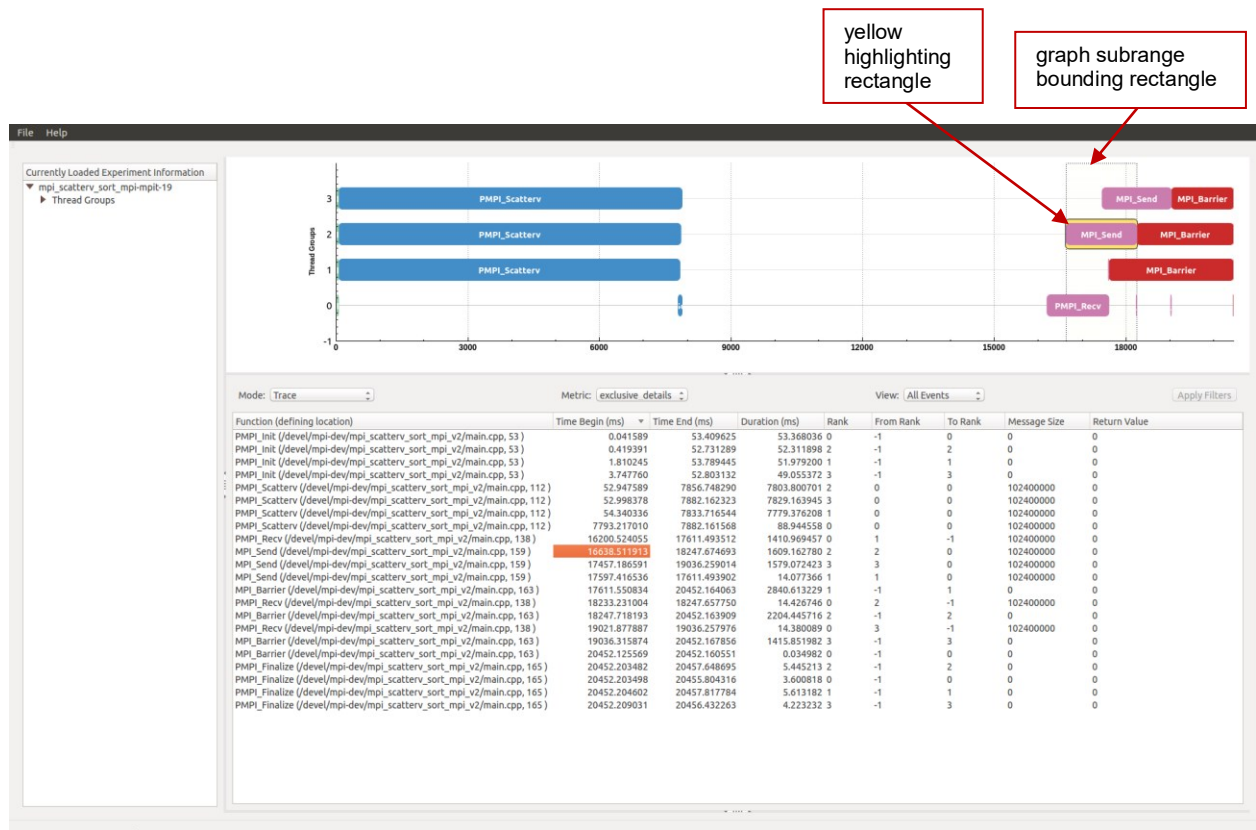


Figure 70 - locating MPI event in graph timeline

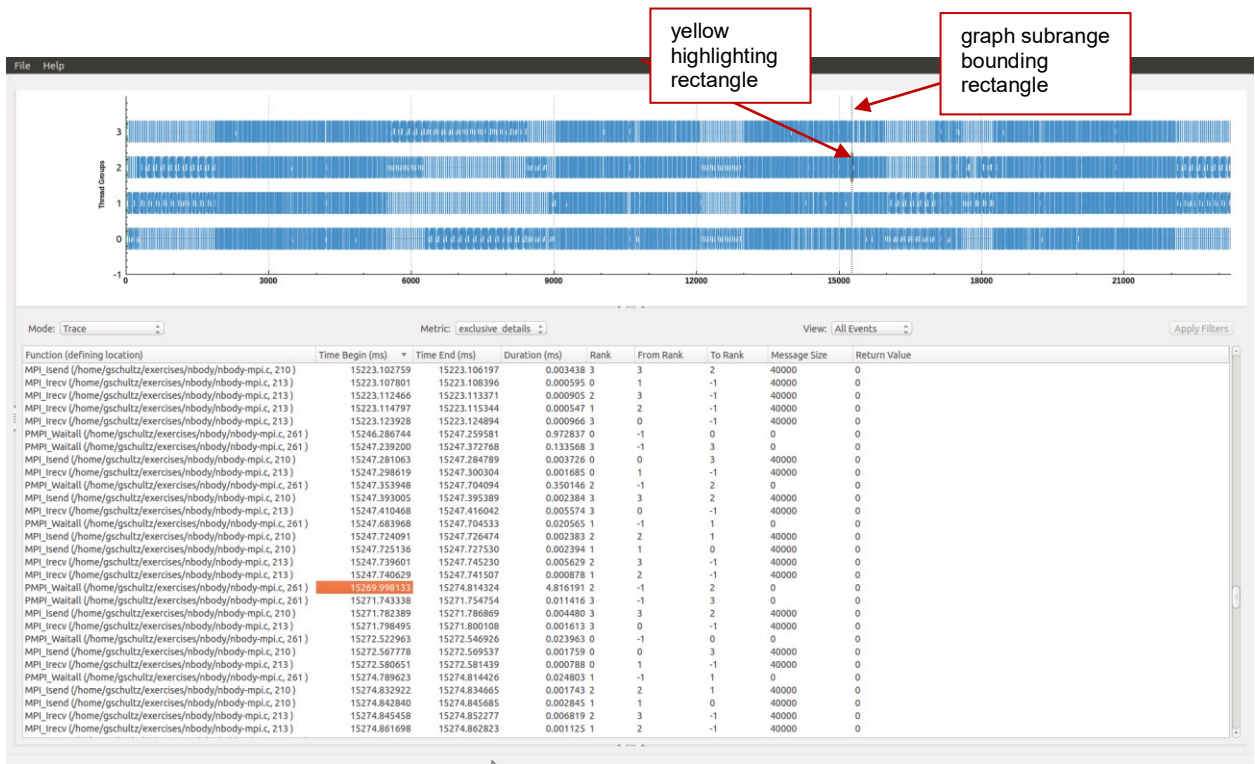


Figure 71 - locating MPI event in a crowded MPI event timeline

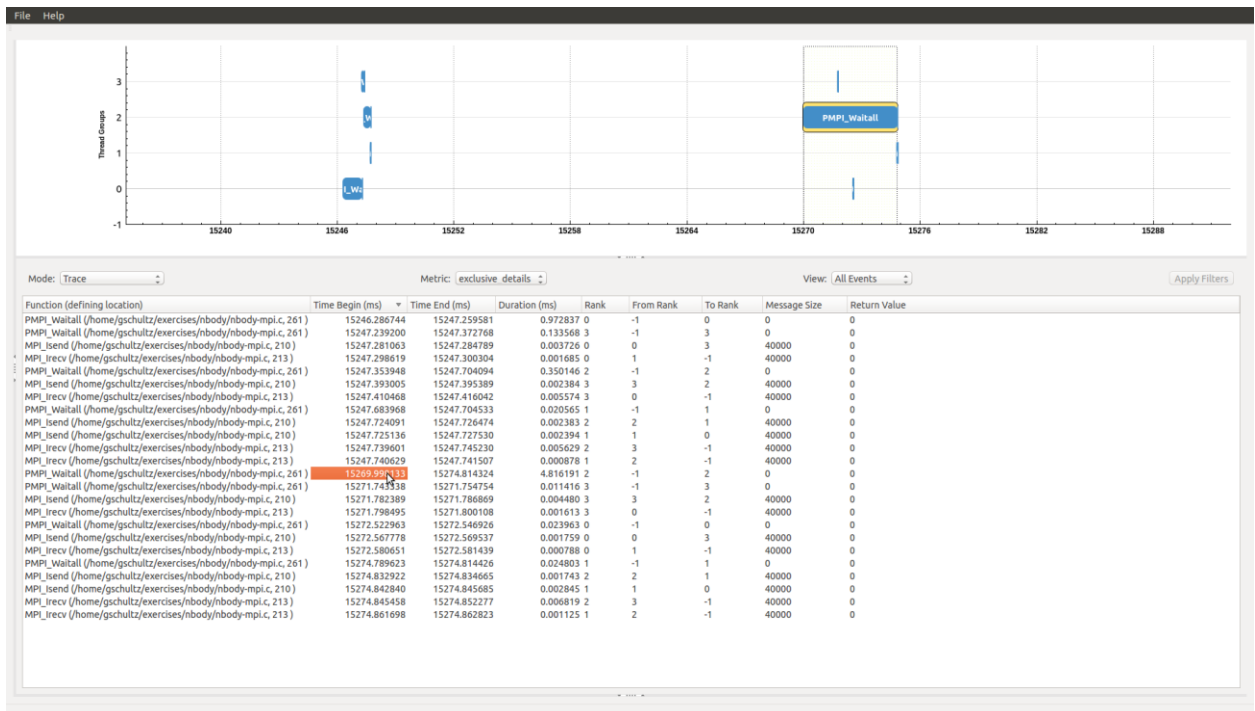


Figure 72 - Using highlighting cues to zoom into selected event

13.6.1.2.13 Using the O|SS GUI to Analyze “pthreads” Experiment Results

Upon loading the “pthreads” experiment the default view appears showing the exclusive time metric values for the functions view. Currently there is no graph generated in the Metric Plot View. However, a calltree graph showing all the caller-callee relationships captured during the experiment execution can be generated and displayed in the Metric Plot View by selecting the “CallTree” option in the “Mode” combo-box (ref Figure 73, “pthreads experiment default view (with calltree graph)”).

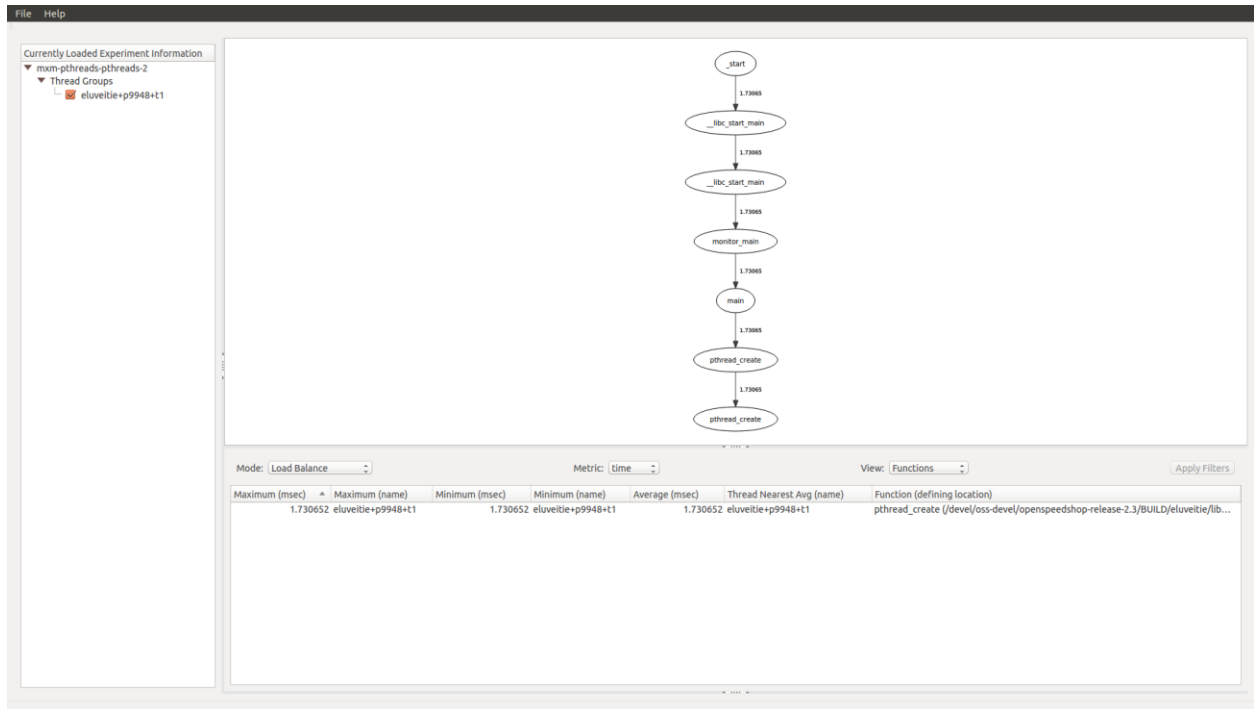


Figure 73 - pthreads experiment default view (with calltree graph)

13.6.1.2.14 Using the O|SS GUI to Analyze Performance of NVIDIA CUDA Applications

To demonstrate how the new GUI can be used to view CPU and GPU activity within an application and generate summary metric results and detailed CUDA event lists two different examples will be discussed.

The default view for the CUDA experiment can be seen in Figure 74. As seen here the user changed the main window configuration to completely close the “Experiment Panel” normally visible on the left-hand side of the main window so that the right-hand panels take the full width of the main window. This is accomplished by using the “handles” in the border area between two panels (ref. the annotation in Figure 74 and Figure 75 for a zoomed in view of the splitter handle between the Metric Plot and Metric Table Views).

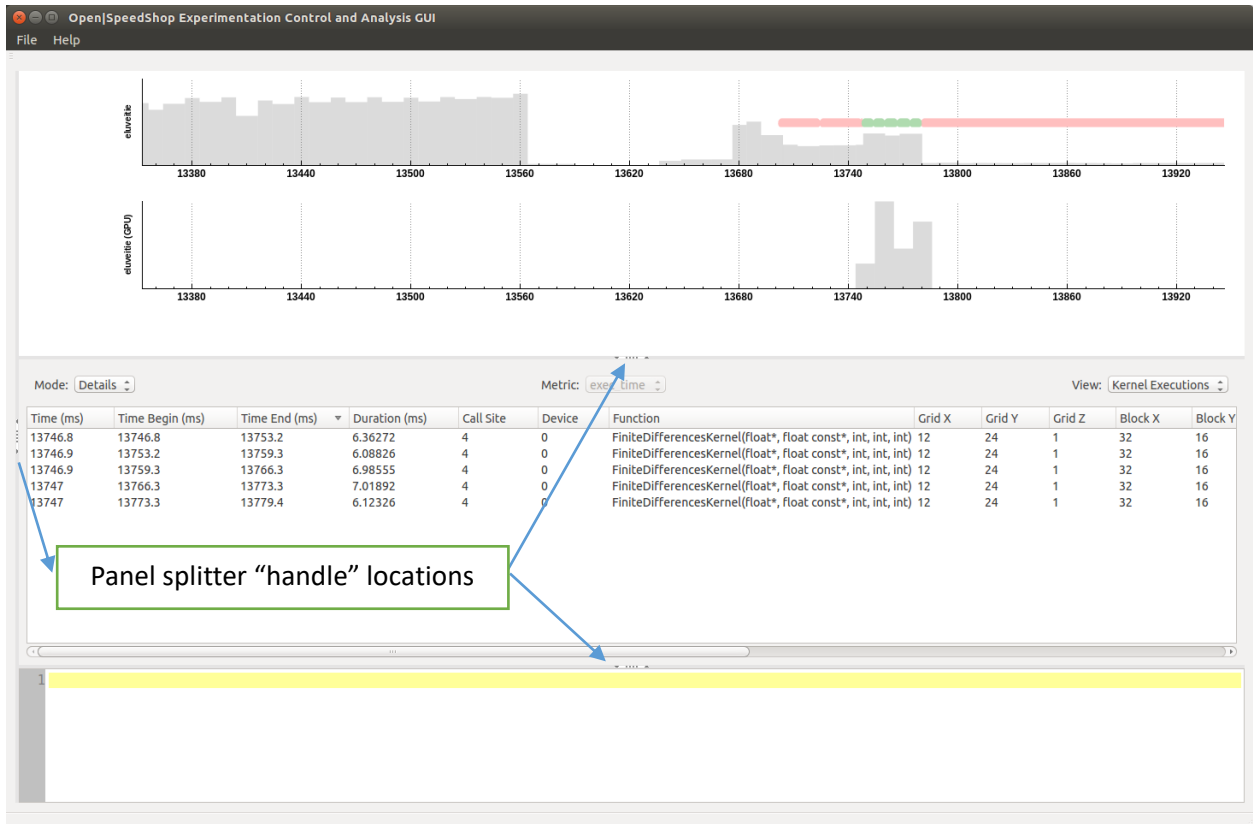


Figure 74 - Default View for the GEMM Experiment



Figure 75 - Zoomed View of Panel Splitter Handles

For the screenshot shown in Figure 76 one can see the CUDA events in the graph timeline. The CUDA events are currently placed on the CPU graph of the CPU + GPU graph view. The rationale for placing them on the CPU graph is so that it does not obstruct the GPU sample counter histogram and the user can clearly see the magnitude of each histogram bar as there should be a direct relationship with CUDA event activity. As discussed previously a red pastel colored rectangle corresponds to a Data Transfer event and a green pastel colored rectangle to a Kernel Execution event. Thus, for the graph shown in Figure 6 there are two Data Transfer events, followed by 5 Kernel Execution events, followed by one Data Transfer event (see annotations on screenshot). There is another annotation linking one of the Kernel Execution events in the Details View to the corresponding graph item in the CUDA timeline. The "Time Begin (ms)" value of the Kernel Execution event will be the x-axis position of the left-edge of the Kernel Execution event rectangle on the graph timeline and the "Time End (ms)" value will be the position of the right edge of the Kernel Execution event rectangle. This screenshot represents the "Details - All Events" view in the area below the Metric Plot View. The additional two screenshots show the "Details - Data Transfers" and "Details - Kernel Executions" views that just contain CUDA Data Transfer or CUDA Kernel Execution events respectively (ref.

Figures 77 and 78).

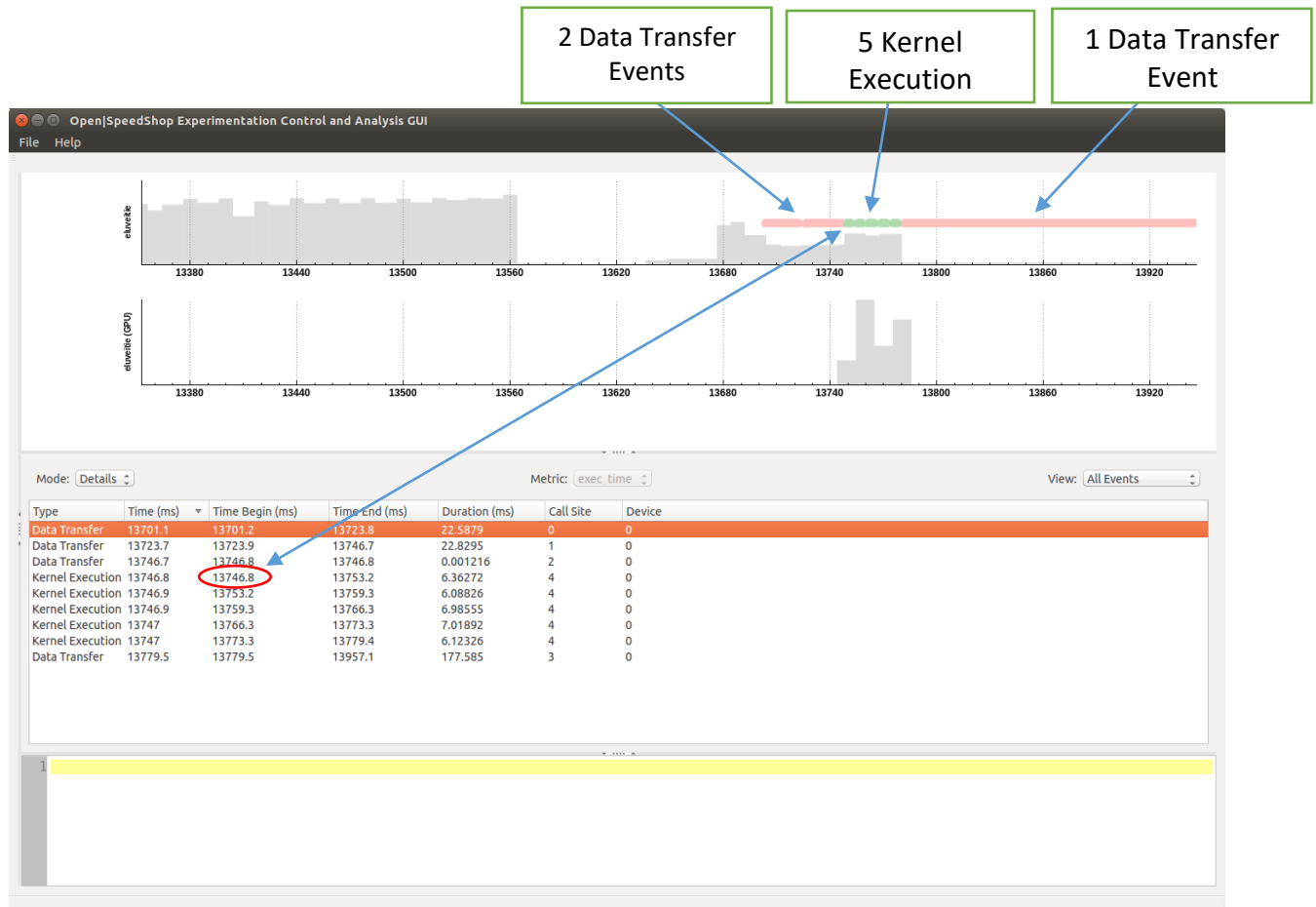


Figure 76 - CUDA Events in Graph Timeline and Details Mode View

For the Data Transfer and Kernel Execution Details views many more columns are displayed showing all the available event information. For the All Events Details view only the common set of event information is shown.

As discussed previously the metric values displayed in the “Metric” mode or the events listed in the various “Details” mode views use the visible time range in the graph timeline as input to the metric computations or filtering logic for which CUDA events to show.

Mode: Details		Metric: exec time								View: Data Transfers	
Time (ms)	Time Begin (ms)	Time End (ms)	Duration (ms)	Call Site	Device	Size	Rate (GB/s)	Kind	Source Kind	Destination Kind	Asynchronous
13701.1	13701.2	13723.8	22.5879	0	0	216 MB	10.0272	HostToDevice	Pageable	Device	false
13723.7	13723.9	13746.7	22.8295	1	0	216 MB	9.92103	HostToDevice	Pageable	Device	false
13746.7	13746.8	13746.8	0.001216	2	0	20 Bytes	0.0164474	HostToDevice	Pageable	Device	false
13779.5	13779.5	13957.1	177.585	3	0	216 MB	1.2754	DeviceToHost	Device	Pageable	false

Figure 77 - Data Transfer Details View

Mode: Details		Metric: exec time								View: Kernel Executions	
Time (ms)	Time Begin (ms)	Time End (ms)	Duration (ms)	Call Site	Device	Function	Grid X	Grid Y	Grid Z	Block X	Block Y
13746.8	13746.8	13753.2	6.36272	4	0	FiniteDifferencesKernel(float*, float const*, int, int, int)	12	24	1	32	16
13746.9	13753.2	13759.3	6.08826	4	0	FiniteDifferencesKernel(float*, float const*, int, int, int)	12	24	1	32	16
13746.9	13759.3	13766.3	6.98555	4	0	FiniteDifferencesKernel(float*, float const*, int, int, int)	12	24	1	32	16
13747	13766.3	13773.3	7.01892	4	0	FiniteDifferencesKernel(float*, float const*, int, int, int)	12	24	1	32	16
13747	13773.3	13779.4	6.12326	4	0	FiniteDifferencesKernel(float*, float const*, int, int, int)	12	24	1	32	16

Figure 78 - Kernel Execution Details View

Another CUDA example will be discussed starting with the performance data collection by running the “ossCUDA” convenience script on a CUDA program which executes several different implementations of matrix multiplication using various performance optimization techniques to demonstrate performance differences, including:

1. Tiling
2. Memory coalescing
3. Avoiding memory bank conflicts
4. Increase floating portion by outer product.
5. Loop unrolling
6. Prefetching

A discussion of the matrix multiplication problem, the various performance optimization techniques used in the application and source-code can be found at <https://sites.google.com/site/5kk70gpu/matrixmul-example>.

```

$ osscuda "./matrixmul"
[openss]: cuda counting all instructions for CPU and GPU.
[openss]: cuda using default periodic sampling rate (10 ms).
[openss]: cuda configuration: "interval=1000000,PAPI_TOT_INS,inst_executed"
Creating topology file for frontend host eluv
Generated topology file: ./cbtfAutoTopology
Running cuda collector.
Program: ./matrixmul
Number of mrnet backends: 1
Topology file used: ./cbtfAutoTopology
executing sequential program: cbtfmun -c cuda --mrnet ./matrixmul
[Matrix Multiply Using CUDA] - Starting...
GPU Device 0: "GeForce GTX 1060" with compute capability 6.1

[CUDA 5632:0] CUPTI_metrics_start(): The selected CUDA device doesn't support continuous GPU event sampling. GPU events
will be sampled at CUDA kernel entry and exit only (not periodically). This also implies CUDA kernel execution will be
serialized, possibly exhibiting different temporal behavior than when executed without performance monitoring.
Naive CPU (Golden Reference)
Processing time: 279.404175 (ms), GFLOPS: 0.360278
threads: x=16 y=16
grid: x=24 y=16
Naive GPU
Processing time: 1.555232 (ms), GFLOPS: 64.725580
Total Errors = 0
Tiling GPU
Processing time: 0.944896 (ms), GFLOPS: 106.533736
Total Errors = 0
Global mem coalescing GPU
Processing time: 1.168640 (ms), GFLOPS: 86.137128
Total Errors = 0
Remove shared mem bank conflict GPU
Processing time: 0.853728 (ms), GFLOPS: 117.910264
Total Errors = 0
Threads perform computation optimization GPU
Processing time: 0.825312 (ms), GFLOPS: 121.969984
Total Errors = 0
Loop unrolling GPU
Processing time: 0.862624 (ms), GFLOPS: 116.694296
Total Errors = 0
Prefetching GPU
Processing time: 1.037664 (ms), GFLOPS: 97.009520
Total Errors = 0
default view for /home/gschultz/Downloads/exercises/cuda/matrixMul/matrixmul-cuda-3.openss
[openss]: The restored experiment identifier is: -x 1
Performance data spans 0.461198 ms from 2017/02/16 23:26:30 to 2017/02/16 23:26:31

Exclusive   % of Exclusive Function (defining location)
Time (ms)   Total   Count
Exclusive
Time
0.605867 32.275192 1 matrixMul_coalescing(float*, float*, float*, int, int) (matrixmul: matrixMul_coalescing.cuh,31)
0.496201 26.433165 1 matrixMul_naive(float*, float*, float*, int, int) (matrixmul: matrixMul_naive.cuh,17)
0.257925 13.739944 1 matrixMul_tiling(float*, float*, float*, int, int) (matrixmul: matrixMul_tiling.cuh,31)
0.211493 11.266461 1 matrixMul_noBankConflict(float*, float*, float*, int, int) (matrixmul:
matrixMul_noBankConflict.cuh,32)
0.108675 5.789235 1 matrixMul_prefetch(float*, float*, float*, int, int) (matrixmul: matrixMul_prefetch.cuh,31)
0.107011 5.700592 1 matrixMul_compOpt(float*, float*, float*, int, int) (matrixmul: matrixMul_compOpt.cuh,31)
0.090019 4.795410 1 matrixMul_unroll(float*, float*, float*, int, int) (matrixmul: matrixMul_unroll.cuh,32)

```

Upon completion of the CUDA experiment the O|SS experiment database will be in the same directory as the profiled application. For this run it is in the file named “matrixmul-cuda-3.openss”. First let’s open the experiment in the O|SS CLI:

opens -cli -f matrixmul-cuda-3.openss

Once the CLI has loaded the experiment the following series of commands are issued to produce metric data:

```
expview -vexec -mexclusive_time,threadmin,threadmax,avg -I432.892:444.104
expview -vxfer -mexclusive_time,threadmin,threadmax,avg -I432.892:444.104
expview -vexec -mexclusive_time,threadmin,threadmax,avg -I441.384:443.981
expview -vxfer -mexclusive_time,threadmin,threadmax,avg -I441.384:443.981
```

The following is a capture of the session:

```
openss>>expview -vexec -mexclusive_time,threadmin,threadmax,avg -I432.892:444.104

Exclusive      Min CUDA      Max CUDA      Average      Function (defining location)
Time (ms)      Kernel       Kernel       Time
                Execution    Execution
                Time Across  Time Across
                ThreadIds (ms) ThreadIds (ms)

0.605867      0.605867      0.605867      0.605867      matrixMul_coalescing(float*, float*,
float*, int, int) (matrixmul: matrixMul_coalescing.cuh,31)
0.496201      0.496201      0.496201      0.496201      matrixMul_naive(float*, float*, float*,
int, int) (matrixmul: matrixMul_naive.cuh,17)
0.257925      0.257925      0.257925      0.257925      matrixMul_tiling(float*, float*, float*,
int, int) (matrixmul: matrixMul_tiling.cuh,31)
0.211493      0.211493      0.211493      0.211493      matrixMul_noBankConflict(float*, float*,
float*, int, int) (matrixmul: matrixMul_noBankConflict.cuh,32)
0.108675      0.108675      0.108675      0.108675      matrixMul_prefetch(float*, float*,
float*, int, int) (matrixmul: matrixMul_prefetch.cuh,31)
0.107011      0.107011      0.107011      0.107011      matrixMul_compOpt(float*, float*, float*,
int, int) (matrixmul: matrixMul_compOpt.cuh,31)
0.090019      0.090019      0.090019      0.090019      matrixMul_unroll(float*, float*, float*,
int, int) (matrixmul: matrixMul_unroll.cuh,32)
openss>>expview -vxfer -mexclusive_time,threadmin,threadmax,avg -I432.892:444.104

Exclusive      Min CUDA      Max CUDA      Average      Function (defining location)
Time (ms)      Data          Data          Time
                Transfer
                Time Across  Time Across
                ThreadIds (ms) ThreadIds (ms)

0.973283      0.973283      0.973283      0.046347      runTest(int, char**) (matrixmul:
matrixMul.cu,163)
openss>>expview -vexec -mexclusive_time,threadmin,threadmax,avg -I441.384:443.981

Exclusive      Min CUDA      Max CUDA      Average      Function (defining location)
Time (ms)      Kernel       Kernel       Time
                Execution    Execution
                Time Across  Time Across
                ThreadIds (ms) ThreadIds (ms)

0.108675      0.108675      0.108675      0.108675      matrixMul_prefetch(float*, float*,
float*, int, int) (matrixmul: matrixMul_prefetch.cuh,31)
0.090019      0.090019      0.090019      0.090019      matrixMul_unroll(float*, float*, float*,
int, int) (matrixmul: matrixMul_unroll.cuh,32)
openss>>expview -vxfer -mexclusive_time,threadmin,threadmax,avg -I441.384:443.981

Exclusive      Min CUDA      Max CUDA      Average      Function (defining location)
Time (ms)      Data          Data          Time
                Transfer
                Time Across  Time Across
                ThreadIds (ms) ThreadIds (ms)
```

Time Across ThreadIds (ms)		Time Across ThreadIds (ms)		
0.287658	0.287658	0.287658	0.047943	runTest(int, char**) (matrixmul: matrixMul.cu,163)

Now let's launch the new GUI automatically loading the same experiment database:

openss-gui -f matrixmul-cuda-3.openss

The series of screenshots shown in Figures 79-82 show the view configuration to achieve the same performance metric results in the GUI as obtained using the CLI.

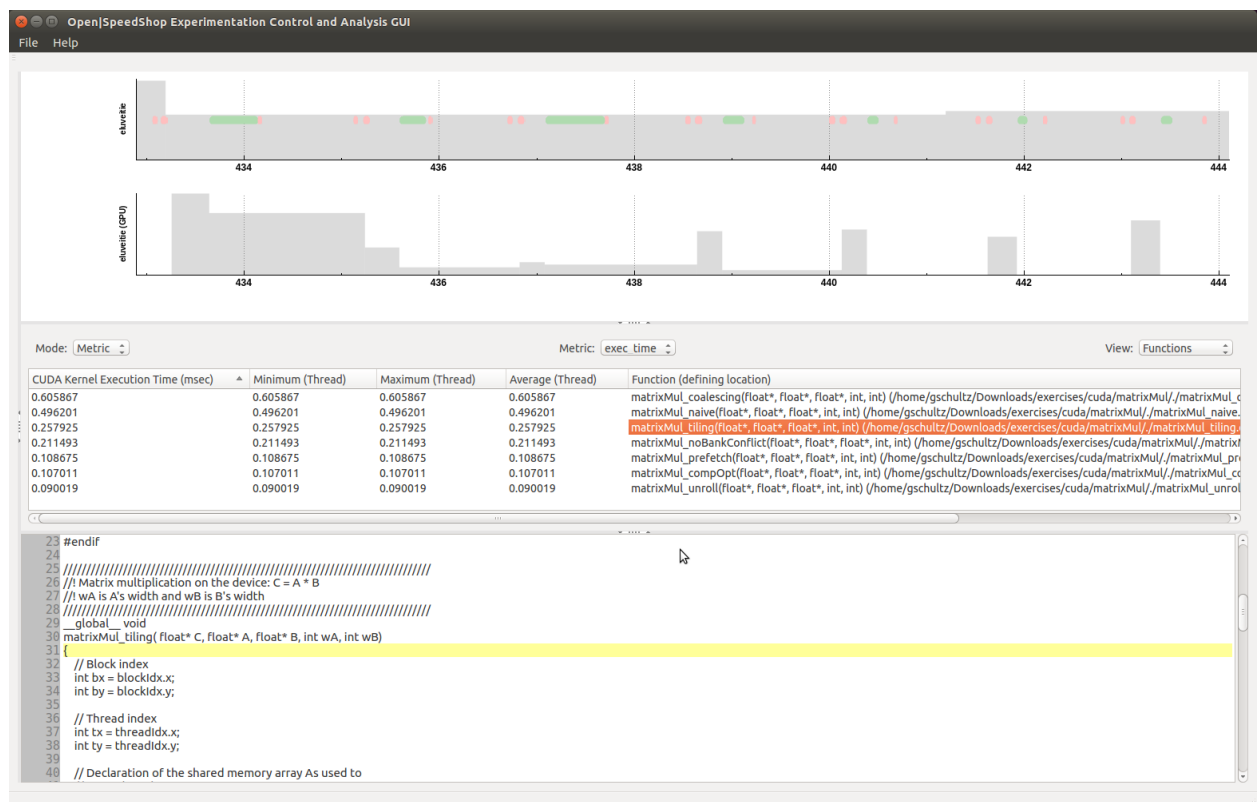


Figure 79 - "expview -vexec -mexclusive_time,threadmin,threadmax,avg -l432.892:444.104"

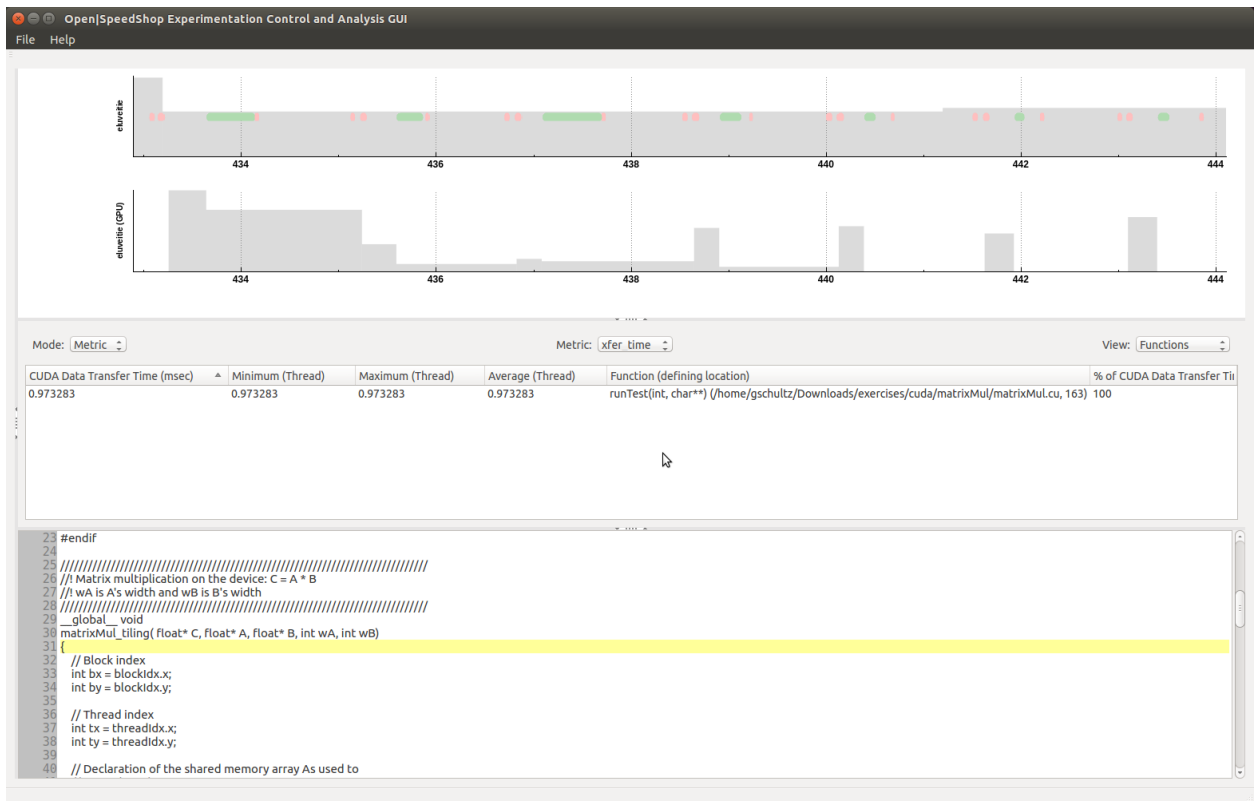


Figure 80 - "expview -vxfer -mexclusive_time,threadmin,threadmax,avg -l432.892:444.104"

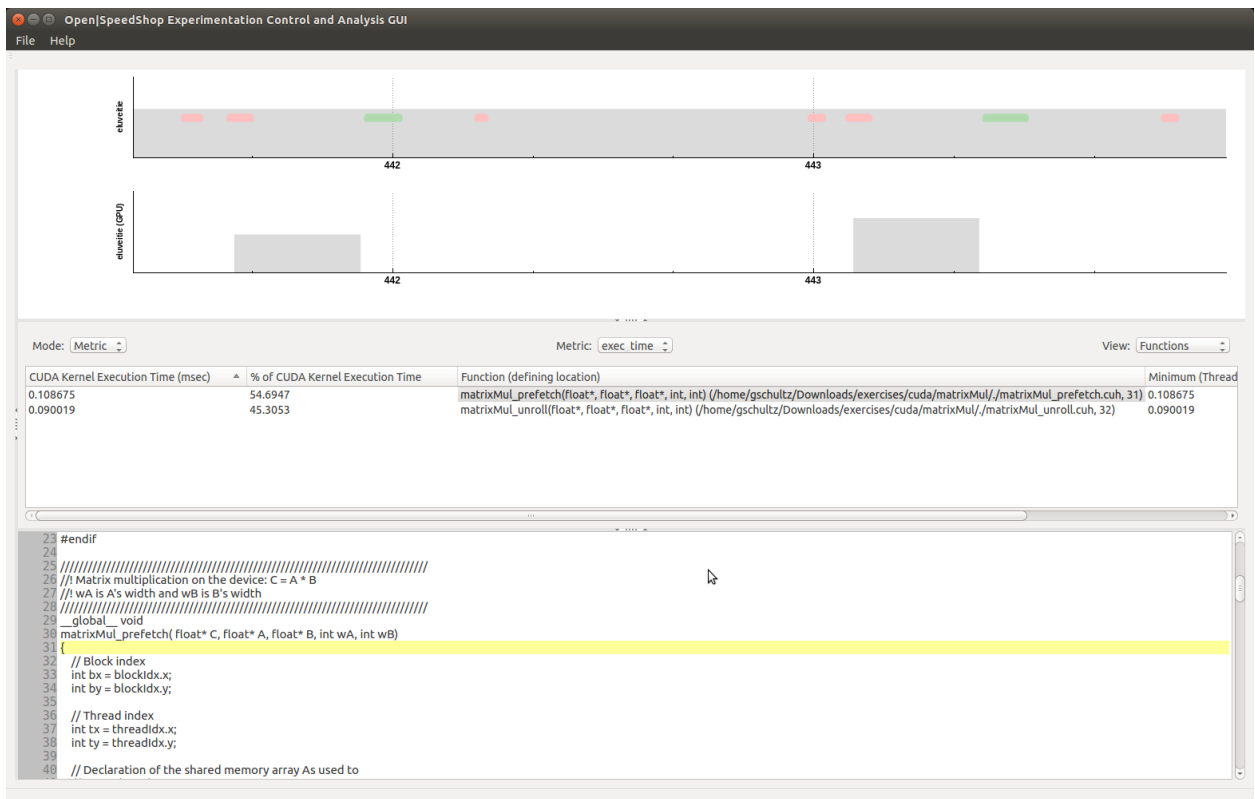


Figure 81 - "expview -vexec -mexclusive_time,threadmin,threadmax,avg -l441.384:443.981"

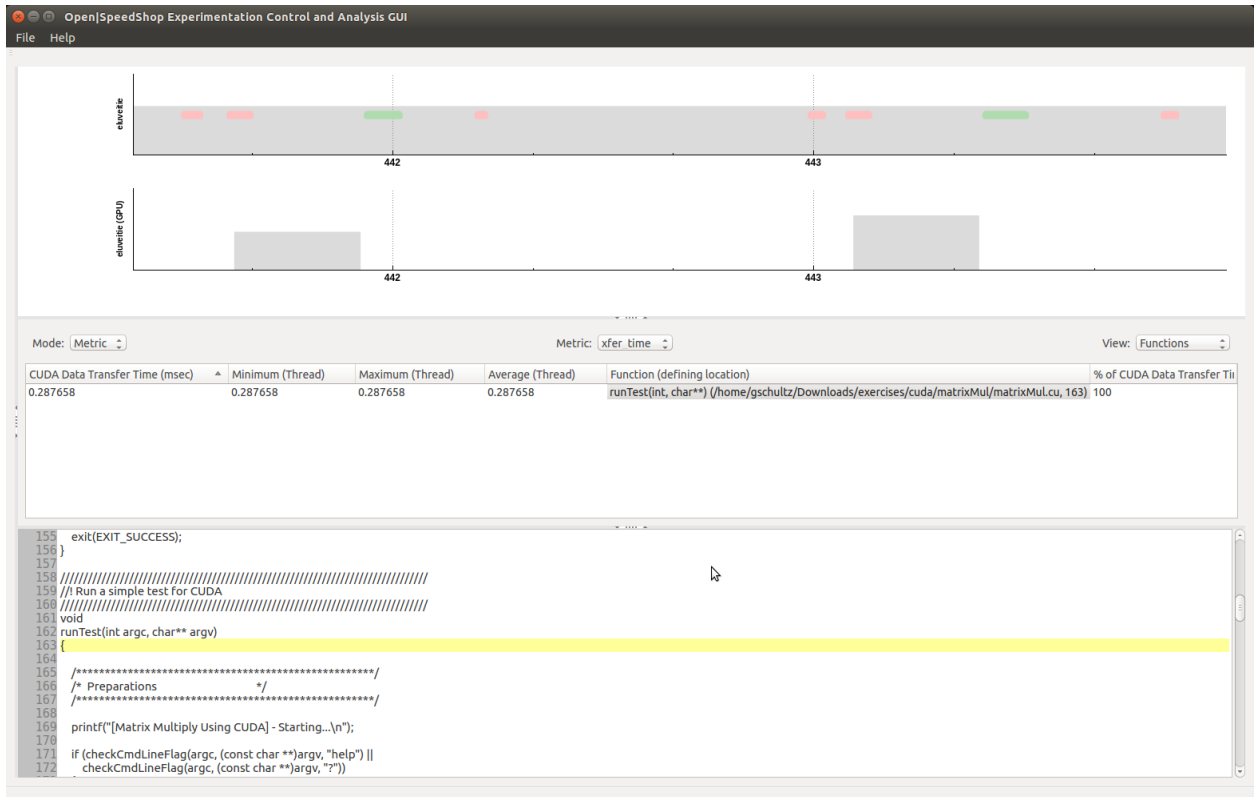


Figure 82 - "expview -vxfer -mexclusive_time,threadmin,threadmax,avg -l441.384:443.981"

Each screenshot caption indicates the corresponding "expview" command in the O|SS CLI.

These screenshots demonstrate that the user can alter the column ordering by holding the left-mouse button when the mouse cursor is over one of the columns and dragging it into a new position. The columns were re-ordered to match the ordering of the CLI views.

14 Special System Support (Static Executables)

14.1 Cray and Blue Gene

The normal mode of operation with respect to running experiments in O|SS doesn't work when the system shared library support is limited. Users must link the collectors into the static executable. O|SS currently has static support on Cray and the Blue Gene P/Q platforms. Users must relink the application with the `osslink` command to add the O|SS collectors and support libraries into their application.

The `osslink` command is a script that will help with linking. Calls to it usually are embedded inside an application's makefile. The user generally needs to locate the makefile target that creates the actual static executable and create a collector target that links in the selected collector. This is an example for re-linking the `smg2000` application:

```
smg2000: smg2000.o
    @echo "Linking" $@ "... "
    ${CC} -o smg2000 smg2000.o ${LFLAGS}

smg2000-pcsamp: smg2000.o
    @echo "Linking" $@ "... "
    osslink -v -c pcsamp ${CC} -o smg2000-pcsamp smg2000.o ${LFLAGS}

smg2000-usertime: smg2000.o
    @echo "Linking" $@ "... "
    osslink -v -c usertime ${CC} -o smg2000-usertime smg2000.o ${LFLAGS}

smg2000-hwcsamp: smg2000.o
    @echo "Linking" $@ "... "
    osslink -v -c hwcsamp ${CC} -o smg2000-hwcsamp smg2000.o ${LFLAGS}

smg2000-io: smg2000.o
    @echo "Linking" $@ "... "
    osslink -v -c io ${CC} -o smg2000-io smg2000.o ${LFLAGS}

smg2000-iot: smg2000.o
    @echo "Linking" $@ "... "
    osslink -v -c iot ${CC} -o smg2000-iot smg2000.o ${LFLAGS}

smg2000-mpi: smg2000.o
    @echo "Linking" $@ "... "
    osslink -v -c mpi ${CC} -o smg2000-mpi smg2000.o ${LFLAGS}
```

Running the re-linked executable will cause the application to write the raw data files to the location that the environment variable `OPENSS_RAWDATA_DIR` specifies. Normally, in the cluster environment in which shared/dynamic executables are run, the conversion from raw data to an O|SS database happens under the hood. However, in this case users must employ the `ossutil` command to manually create the database file. Of course, users can add the `ossutil` command to a batch script to

eliminate manually issuing the command. Once the O|SS database files are created, users can view them normally with the GUI or CLI.

Here's an example of a job script that will execute these steps:

```
#PBS -q debug
#PBS -N smg2000-pcsamp
...
# must have a clean raw data directory each run
rm -rf /home/USER/smg2000/test/raw
mkdir /home/USER/smg2000/test/raw

setenv OPENSS_RAWDATA_DIR /home/USER/smg2000/test/raw
setenv OPENSS_DB_DIR /home/USER/smg2000/test/

cd /home/jgalaro/smg2000/test

# needs -b to have the original executable path available and match where
# the application was run when doing ossutil
aprun -b -n 16 /home/USER/smg2000/test/smg2000-pcsamp

# creates a X.0.openss database file, please
# load the module pointing to openspeedshop before accessing ossutil
ossutil /home/jgalaro/smg2000/test/raw
```

The executable path that is used to process symbols after the run is complete must match where the executable was run. The executable path must match the path in the raw data that is written to the directory that `OPENSS_RAWDATA_DIR` represents. If the aprun “-b” option is not used, then the executable is run in a temporary system directory and the raw data reflects that directory path for the executable instead of the path where the executable is located when the job is initiated. As a result, ossutil will be unable to resolve the symbols.

There have been recent changes to the shared library support in O|SS. Dynamic shared library support is now available in newer Cray and Blue Gene operating systems. There is support for both shared and static binaries on the Cray and Blue Gene Q platforms.

14.1.1 osslink Command Information

The osslink command links the O|SS collectors and runtime libraries into the static executable and manages setting the appropriate libraries based on the collector value input to the command. Here is the help output for osslink:

```
osslink --help

Usage: /opt/ossctbf_cmake_only_july10/bin/osslink -c collector [options] compiler file ...

-h, --help
```



```
-c, --collector <collector name>
```

Where collector is the name of the O|SS collector to link into the application. See the openss man page for a description of the available experiments provided by O|SS. This is a mandatory option.

```
-i | --mpitype
```

For MPI experiments, set the OPENSS_MPI_IMPLEMENTATION value to the MPI implementation specified. Valid options are:

```
mpich
mpich2
mvapich
mvapich2
openmpi
mpt
lam
lampi
```

```
-v, --verbose
```

14.1.2 Cray-Specific Static aprun Information

Note: In the above execution of the statically linked executable, the `-b` option must be added to the `aprun` call. The option is needed because O|SS stores information about the executable location when it is running. Without the `-b` option the executable is run in a temporary location that is unavailable when the raw data information is converted into the O|SS database file.

14.1.3 Changing parameters to the experiments

Note: When running the statically linked executable with the O|SS collectors linked in, the workflow is different. Since the more flexible convenience scripts can't be used, users must set environment variables to change the arguments to the experiments.

Examples of the environment variables that can be changed are as follows:

Environment Variable	Represents	Experiment Type
OPENSS_PCSAMP_RATE	Sampling Rate.	pcsamp
OPENSS_USERTIME_RATE	Sampling Rate.	usertime
OPENSS_HWC_EVENT	PAPI or Native Event Name.	hwc
OPENSS_HWC_THRESHOLD	How many event occurrences before sample taken.	hwc
OPENSS_HWCSAMP_EVENTS	List of PAPI or Native Event Names.	hwcsamp
OPENSS_HWCSAMP_RATE	Sampling Rate.	hwcsamp

OPENSS_HWCTIME_EVENT	PAPI or Native Event Name.	hwctime
OPENSS_HWCTIME_THRESHOLD	How many event occurrences before sample taken.	hwctime
OPENSS_IO_TRACED	List of I/O functions to collect data for.	io
OPENSS_IOT_TRACED	List of I/O functions to collect data for.	iot
OPENSS_MPI_TRACED	List of MPI functions to collect data for.	mpi
OPENSS_MPIT_TRACED	List of MPI functions to collect data for.	mpit

15 Setup and Build for O|SS

O|SS is set up to work with a variety of processor types, including Intel, AMD, Intel Phi, PPC and ARM architectures. It has been tested on many Linux Distributions, including SLES, SUSE, RHEL, Fedora Core, CentOS, Debian, Ubuntu and many others. It has been installed on IBM Blue Gene and Cray systems. The O|SS website contains information on special builds and usage instructions.

Build information can be found on the O|SS website:

<http://www.openspeedshop.org>

Build guidelines are described in the following sections. We recommend using spack to build O|SS if at all possible. It is the easiest, cleanest method compared to the previous install-tool mechanism. Spack also creates the runtime environment module file as well. This makes it much easier to build and use O|SS.

15.1 Installing O|SS with Spack

Spack is a multi-platform package manager that builds and installs multiple versions and configurations of software. It works on Linux, macOS, and many supercomputers. Spack is non-destructive: installing a new version of a package does not break existing installations, so many configurations of the same package can coexist. Most importantly, Spack is *simple*. It offers a simple *spec* syntax so that users can specify versions and configuration options concisely. Spack is also simple for package authors: package files are written in pure Python, and specs allow package authors to maintain a single file for many different builds of the same package. If you're new to spack and want to start using it, see [Getting Started](#), or refer to the full manual below.

O|SS can be built with spack by downloading the spack package source, as described in the O|SS Spack Build Guide. It is possible to build O|SS with one spack install command. Spack will download all the dependent packages that O|SS needs, build and install them. After the dependent packages are built, spack will download, build, and install O|SS. Spack creates module file for all the packages that it builds. The instructions below identify where to find the module file and how to load it.

Spack allows for enabling and disabling O|SS optional build arguments/parameters via a spack feature named variants. For O|SS, the main variants are MPI implementation identifiers which are used to build the MPI collectors for those MPI implementations. The spack build command:

```
spack install openspeedshop +openmpi +mvapich2
```

will build O|SS with all the collectors, including MPI collectors that will work on OpenMPI and Mvapich2 based applications. Without those variants, all the non-mpi specific experiments will be built, but not the MPI collectors (no ossmpi, ossmpip, and ossmpit support).

Please use the install information in the O|SS Spack Build guide, which can be found on this webpage: <https://openspeedshop.org/documentation>

15.2 Installing O|SS with the install-tool command

O|SS comes with a set of bash install scripts that will build O|SS and any components it needs from source tarballs. First it will check to see if the correct supporting software is installed on the system. If the needed software isn't installed, it will ask to build it for the user. The only thing users must do is provide a few arguments for the install script. For a normal setup, just specify the directory to install in, what build task is desired and the location of the MPI and QT installs. For example:

Build only the krell-root

```
./install-tool --build-krell-root
--krell-root-prefix /opt/krellroot_v2.4.0
--with-openmpi /opt/openmpi-1.8.2
```

Build cbtf components using the krell-root

```
./install-tool --build-cbtf-all
--cbtf-prefix /opt/cbtf_only_v2.4.0
--krell-root-prefix /opt/krellroot_v2.4.0
--with-openmpi /opt/openmpi-1.8.2
--with-cupti /usr/local/cuda-6.5/extras/CUPTI
--with-cuda /usr/local/cuda-6.5
```

Build only OSS using the cbtf components and the krell-root

```
./install-tool --build-oss
--cbtf-prefix /opt/cbtf_only_v2.4.0
--krell-root-prefix /opt/krellroot_v2.4.0
--openss-prefix /opt/osscbtf_v2.4.0
--with-openmpi /opt/openmpi-1.8.2
--with-cupti /usr/local/cuda-6.5/extras/CUPTI
--with-cuda /usr/local/cuda-6.5
```

After the install has successfully completed, a few important environment variables must be set. Set a variable for the install location, so it can be reused. If O|SS was installed with more than one MPI version, specify which to use with `OPENSS_MPI_IMPLEMENTATION`. Lastly, add the O|SS and Krell externals (root) bin directory to your `PATH` and add lib64 directories to your `LD_LIBRARY_PATH`. See the sections below for examples of the necessary environment variables that must be set.

15.3 Execution Runtime Environment Setup

If O|SS was built with Spack, the runtime environment setup files are generated for you. Just load the module file created for O|SS and your environment is initialized for use.

If using the install-tool mechanism, then this section gives an example of a module file, softenv file and dotkit that can be used to set up the O|SS execution environments.

NOTE: For versions 2.3.0 and beyond the old O|SS module files will need updating because these versions now use a multicast network that is incorporated into the Component Based Tool Framework (CBTF) components. The new module file needs new settings to set up and operate the CBTF components and the multicast network. New module example files are listed below.

Also: For builds of O|SS done with a compiler installed in a non-standard location (a module was loaded for the compiler), please set up the library path to that compiler's libraries in the O|SS module file. See an example below.

15.3.1 Example module file

Here is an example of a module file used for a cluster installation. Use module load <filename of module file> to activate the O|SS runtime environment:

```
##%Module1.0#####
##
##
## openss modulefile
##
proc ModulesHelp { } {
    global version openss

    puts stderr "\ntopenss - loads the OpenSpeedShop software & application environment"
    puts stderr "\n\tThis adds $oss/* to several of the"
    puts stderr "\tenvironment variables."
    puts stderr "\n\tVersion $version\n"
}

# NOTE -----
# The paths may need adjustment for different library naming schemes
# NOTE -----
#

module-whatis "Loads the OpenSpeedShop runtime environment."

# for Tcl script use only
set version 2.4.0.latest

# Set up variables to reference later for the krell root, cbtf, and OpenSpeedShop proper
set base /home/fred/openss/power
```

```

set root      ${base}/krellroot_v2.4.0.latest
set cbtf      ${base}/cbtf_v2.4.0.latest
set cbtfk     ${base}/cbtf_v2.4.0.latest
set oss       ${base}/osscbtf_v2.4.0.latest
set qtgraph   ${base}/QtGraph-1.0.0
set graphviz  ${base}/graphviz-2.41.0

# XPLAT_RSH is needed for MRNet which is now needed for use in CBTF
setenv XPLAT_RSH ssh

# For the mpi experiments only - specify the MPI implementation of your
# application that will be run with OpenSpeedShop. These are the
# mpi, mpit, and mpip experiments. All other experiment types will
# ignore this setting. It is only needed for mpi, mpit, and mpip.
setenv CBTF_MPI_IMPLEMENTATION openmpi
setenv OPENSS_MPI_IMPLEMENTATION openmpi

# This is needed if you use the --offline argument following the
# convenience scripts, for example: osspcsamp --offline "mpirun -np 4 ./nbody"
# This is the offline mode of operation which is now built into the
# CBTF based version of OpenSpeedShop
setenv OPENSS_RAWDATA_DIR .

# Only need these CBTF specific variables for situations where the environment is not passed
setenv MRNET_COMM_PATH $cbtfk/sbin/cbtf_mrnet_commnode
setenv CBTF_MRNET_BACKEND_PATH $cbtfk/sbin/cbtf_libcbtf_mrnet_backend

# Set up the paths for the OSS/CBTF version of OpenSpeedShop
prepend-path PATH $root/bin
prepend-path PATH $cbtf/bin
prepend-path PATH $cbtfk/sbin
prepend-path PATH $cbtfk/bin
prepend-path PATH $oss/bin
prepend-path MANPATH $oss/share/man

# Set up the dyninst runtime library path for the OSS/CBTF version of OpenSpeedShop
# This is required for finding loops and gathering symbol table information.
setenv DYNINSTAPI_RT_LIB $root/lib/libdyninstAPI_RT.so

# Set up the library paths for the OSS/CBTF version of OpenSpeedShop
prepend-path LD_LIBRARY_PATH $root/lib64
prepend-path LD_LIBRARY_PATH $root/lib
prepend-path LD_LIBRARY_PATH $cbtf/lib64
prepend-path LD_LIBRARY_PATH $cbtfk/lib64
prepend-path LD_LIBRARY_PATH $oss/lib64
prepend-path LD_LIBRARY_PATH $qtgraph/lib/5.6.1
prepend-path LD_LIBRARY_PATH $graphviz/lib
prepend-path LD_LIBRARY_PATH /usr/local/cuda-8.0/extras/CUPTI/lib64

# Set up the python path so that the python scripting API can find
# the openss python module files.
setenv PYTHONPATH $oss/lib64/openspeedshop

```

Here is an example module file for a Cray installation:

```

##%Module1.0#####
##
##
## oss cbtf 2.4.0 modulefile
##
proc ModulesHelp { } {

```

```

global version openspeedshop-cbtf

puts stderr "\topenspeedshop-cbtf - Loads the OpenSpeedShop software target back-end (be) and front-end
(fe) execution environment for Cray"
puts stderr "\n\tVersion $version\n"
}

module-whatis "Loads the OpenSpeedShop target back-end node (be) and front-end (fe) execution
environment."

# for Tcl script use only
set version 2.4.0
set root_prefix /p/home/galarowi/openss/krellroot_v2.4.0
set cbtf_prefix /p/home/galarowi/openss/cbtf_v2.4.0
set oss_prefix /p/home/galarowi/openss/ossbtf_v2.4.0
# Path to the qt3 toolkit
set qt /p/home/galarowi/openss/krellroot_v2.4.0/qt3
# Path to the libraries needed for the qt4/qt5 toolkit needed for the new Qt4/Qt5 based gui
set graphviz /p/home/galarowi/openss/graphviz-2.40.1
set qtgraph /p/home/galarowi/openss/QtGraph-1.0.0
#set papi /opt/cray/papi/5.4.3.1

setenv OPENSS_DOC_DIR $oss_prefix/share/doc/packages/OpenSpeedShop

# This is needed if you use the --offline argument following the
# convenience scripts, for example: osspcsamp --offline "mpirun -np 4 ./nbody"
# This is the offline mode of operation which is now built into the
# CBTF based version of OpenSpeedShop
setenv OPENSS_RAWDATA_DIR .

# For the mpi experiments only - specify the MPI implementation of your
# application that will be run with OpenSpeedShop. These are the
# mpi, mpit, and mpip experiments. All other experiment types will
# ignore this setting. It is only needed for mpi, mpit, and mpip.
setenv CBTF_MPI_IMPLEMENTATION mpich
setenv OPENSS_MPI_IMPLEMENTATION mpich

# XPLAT_RSH is needed for MRNet which is now needed for use in CBTF
setenv XPLAT_RSH ssh

# Only need these CBTF specific variables for situations where the environment is not passed
setenv MRNET_COMM_PATH $cbtf_prefix/sbin/cbtf_mrnet_commnode
setenv CBTF_MRNET_BACKEND_PATH $cbtf_prefix/sbin/cbtf_libcbtf_mrnet_backend

# oss_prefix_target/bin must come first to
# find the osslink in the target directory

#prepend-path PATH $papi/bin
prepend-path PATH $root_prefix/bin
prepend-path PATH $oss_prefix/bin
prepend-path PATH $cbtf_prefix/bin
prepend-path PATH $cbtf_prefix/sbin
prepend-path MANPATH $oss_prefix/share/man

eval set [ array get env HOME ]
set ownmoddir $HOME/privatemodules

# Set up the dyninst runtime library path for the OSS/CBTF version of OpenSpeedShop
# This is required for finding loops and gathering symbol table information.
setenv DYNINSTAPI_RT_LIB $root_prefix/lib64/libdyninstAPI_RT.so

```

```
# Might need this if you use the system installed papi, here as a hint
#prepend-path LD_LIBRARY_PATH $papi/lib64

# Setup the library paths for the runtime environment
prepend-path LD_LIBRARY_PATH $root_prefix/lib
prepend-path LD_LIBRARY_PATH $root_prefix/lib64
prepend-path LD_LIBRARY_PATH $cbtf_prefix/lib64
prepend-path LD_LIBRARY_PATH $oss_prefix/lib64

# Setup the library paths for the qt3 runtime environment
prepend-path LD_LIBRARY_PATH $qt/lib64
# Setup the library paths for the libraries needed for the qt4/qt5 new GUI runtime environment
prepend-path LD_LIBRARY_PATH $graphviz/lib
prepend-path LD_LIBRARY_PATH $qtgraph/lib64/4.8.6
```

15.3.2 Example softenv file

This is an example of a softenv file used for a O|SS offline version Blue Gene/Q installation. Use the “resoft <filename of softenv file>” command to activate the O|SS runtime environment:

```
# The O|SS .soft file.
# Remember to type "resoft" after working on this file.

OSS = /home/projects/oss/oss
KROOT = /home/projects/krellroot
TARCH = bgq

# Set up OSS environment variables

# Find the executable portions of O|SS (order is important here)
PATH += $KROOT/$TARCH/bin
PATH += $KROOT/bin
PATH += $OSS/$TARCH/bin
PATH += $OSS/bin

# Find the libraries for O|SS (order is important here)
LD_LIBRARY_PATH += $KROOT/$TARCH/lib64
LD_LIBRARY_PATH += $KROOT/lib64
LD_LIBRARY_PATH += $KROOT/lib
LD_LIBRARY_PATH += $OSS/$TARCH/lib64
LD_LIBRARY_PATH += $OSS/lib64

# Find the runtime collectors
OPENSS_PLUGIN_PATH = $OSS/$TARCH/lib64/openspeedshop

# Find Dyninst for generation of per-loop statistics
DYNINSTAPI_RT_LIB $KROOT/lib64/libdyninstAPI_RT.so

# Tell the tool what the application MPI implementation is
# Needed if supporting multiple implementations and running the "mpi", "mpit", or "mpiof" experiments
OPENSS_MPI_IMPLEMENTATION = mpich2

# Paths to documentation and man pages
OPENSS_DOC_DIR = $OSS/share/doc/packages/OpenSpeedShop
```



```
MANPATH = $OSS/share/man
```

```
# Use the basic environment.  
@default
```

15.3.3 Example dotkit file

Here is an example of a dotkit file used for a 64-bit cluster platform original offline version OJSS installation, where all components were installed into the same prefix. It is not generalized to support platforms other than the 64-bit cluster it was written for. Use the “use <filename of dotkit file>” command to activate the OJSS runtime environment. Note: Do not include the “.dk” portion of the filename when using the “use” command.

```
#c performance/profile  
#d OJSS (Version 2.4.0)  
dk_setenv OPENSS /usr/global/tools/openspeedshop/oss-dev/OSS_2.4.0  
dk_setenv KROOT /usr/global/tools/openspeedshop/oss-dev/krellroot_2.4.0  
dk_setenv CBTF /usr/global/tools/openspeedshop/oss-dev/cbtf_2.4.0  
# XPLAT_RSH is needed for MRNet which is now needed for use in CBTF  
dk_setenv XPLAT_RSH ssh  
# If cuda present, then we need hooks to the CUPTI interface  
dk_setenv CUDA_PATH /usr/global/tools/cuda-8.0  
# If the new Qt4/Qt5 GUI was built, we the paths to graphviz and QtGraph  
dk_setenv QTGRAPH /usr/global/tools/openspeedshop/oss-dev/QtGraph-1.0.0  
dk_setenv GRAPHVIZ /usr/global/tools/openspeedshop/oss-dev/graphviz-2.40.1  
  
# For the mpi experiments only - specify the MPI implementation of your  
# application that will be run with OpenSpeedShop. These are the  
# mpi, mpit, and mpip experiments. All other experiment types will  
# ignore this setting. It is only needed for mpi, mpit, and mpip.  
dk_setenv OPENSS_MPI_IMPLEMENTATION mvapich2  
  
dk_setenv OPENSS_PLUGIN_PATH $OPENSS/lib64/openspeedshop  
dk_setenv OPENSS_DOC $OPENSS/share/doc/packages/OpenSpeedShop/  
  
# Find Dyninst for generation of per-loop statistics  
dk_setenv DYNINSTAPI_RT_LIB $KROOT/lib64/libdyninstAPI_RT.so  
  
dk_alter PATH $KROOT/bin  
dk_alter PATH $OPENSS/bin  
  
dk_alter PATH $CBTF/bin  
dk_alter PATH $CBTF/sbin  
dk_alter LD_LIBRARY_PATH $CBTF/lib64  
dk_alter LD_LIBRARY_PATH $KROOT/lib64  
dk_alter LD_LIBRARY_PATH $KROOT/lib  
  
dk_alter LD_LIBRARY_PATH $OPENSS/lib64  
dk_alter LD_LIBRARY_PATH $QTGRAPH/lib/5.6.1  
dk_alter LD_LIBRARY_PATH $GRAPHVIZ/lib  
dk_alter LD_LIBRARY_PATH $CUDA_PATH/extras/CUPTI/lib64
```

16 Additional Information and Documentation Sources

16.1 Final Experiment Overview

In the table below we match up a few general questions users may ask with the experiments they may want to run to find answers.

Where does my code spend most of its time?
<ul style="list-style-type: none">• Flat profiles (pcsamp)• Getting inclusive/exclusive timings with call paths (usertime)• Identifying hot call paths (usertime + HP analysis)
How do I analyze cache performance?
<ul style="list-style-type: none">• Measure memory performance using hardware counters (hwc)• Compare to flat profiles (custom comparison)• Compare multiple hardware counters (N x hwc, hwcsamp)
How to identify I/O problems?
<ul style="list-style-type: none">• Study time spent in I/O routines (io, iot and lightweight iop)• Compare runs under different scenarios (custom comparisons)
How to identify memory problems?
<ul style="list-style-type: none">• Study time spent in memory allocation/de-allocation routines (mem)• Look for load imbalance (LB view) and outliers (CA view)
How do I find parallel inefficiencies in OpenMP and/or threaded applications?
<ul style="list-style-type: none">• Study time spent in POSIX thread routines (pthreads)• Look for load imbalance (LB view) and outliers (CA view)
How do I find parallel inefficiencies in MPI applications?
<ul style="list-style-type: none">• Study time spent in MPI routines (mpi, mpit, and lightweight mpip)• Look for load imbalance (LB view) and outliers (CA view)
How do I find parallel inefficiencies in NVIDIA CUDA applications?
<ul style="list-style-type: none">• Study time spent in CUDA routines and the CUDA event execution trace. (cuda)

16.2 Additional Documentation

The Python scripting API documentation can be found at http://www.openspeedshop.org/docs/pyscripting_doc or in the `.../share/doc/packages/openspeedshop/pyscripting_doc` folder in the install directory.

There also are man pages for openss and every convenience script. There's also a quick-start guide available for download from <http://www.openspeedshop.org>.

There is an O|SS Forum-type email alias where users can ask questions and read previous posts: oss-questions@openspeedshop.org. Use this URL to sign up: <https://groups.google.com/a/krellinst.org/forum/?hl=en-!forum/oss-questions>

There also is an email list to which users can send questions without joining the group: oss-contact@openspeedshop.org.

17 Convenience Script Basic Usage Reference Information

This section provides a quick overview of the convenience scripts available to either compare experiment data with other experiment data or to gather performance information for each of the various performance metric types that O|SS supports.

17.1 Suggested Workflow

We recommend an O|SS workflow consisting of two phases: gathering the performance data using the convenience scripts, then using the GUI or CLI to view the data.

17.2 Convenience Scripts

Users are encouraged to employ the convenience scripts (for dynamically linked applications) that hide some of the underlying options for running experiments. The full command syntax can be found in the User's Guide. The script names correspond to the experiment types and are: **osspsamp**, **ossusertime**, **osshwc**, **osshwcsamp**, **osshwctime**, **ossio**, **ossiot**, **ossmipi**, **ossmipit**, **ossiop**, **ossmem**, **ossmptp**, **osspthreads**, **ossmip** and **osscuda**, plus an **osscompare** script. Note: If using the offline operating mode, be sure to set **OPENSS_RAWDATA_DIR**. (See **KEY ENVIRONMENT VARIABLES** section for information.)

O|SS no longer gathers loop information by default, the "--loops" option is required for the loop level information to be gathered and subsequently viewed. For example: **osspsamp --loops "mpirun -np 256 ./smg2000 -n 5 5 5"**

O|SS will gather information about the vector instructions that were executed in the application run, provided that a sample was taken at the address that corresponds to a vector instruction. There are three options that will enable this feature:

- **--vinstr128** Find vector instructions with operand sizes that are 128 bits or greater
- **--vinstr256** Find vector instructions with operand sizes that are 256 bits or greater
- **--vinstr512** Find vector instructions with operand sizes that are 512 bits or greater

For example: **osspsamp -vinstr512 "mpirun -np 256 ./smg2000 -n 5 5 5"**

When running O|SS, use the same syntax that is used to run the application/executable outside of O|SS, but enclosed in quotes; for example:

Using MPI drivers like mpirun:

osspsamp "mpirun -np 512 ./smg2000 -n 5 5 5"

Using SLURM/srun:

osspsamp "srun -N 64 -n 512 ./smg2000 -n 5 5 5"

Redirection to/from files inside quotes can be problematic. See the convenience script "man pages for more information.

17.3 Report and Database Creation

Running the pcsamp experiment on the sequential program named mexe, with the command:

```
osspsamp mexe
```

results in a default report and the creation of a SQLite database file:

```
mexe-pcsamp.openss
```

in the current directory.

Here's the report:

% CPU Time	CPU time	Function
48.990	11.650	f3 (mexe: m.c, 24)
33.478	7.960	f2 (mexe: m.c,15)
17.451	4.150	f1 (mexe: m.c,6)
0.084	0.020	work(mexe:m.c,33)

To access alternative views in the GUI, use the command:

```
openss -f mexe-pcsamp.openss
```

to load the database file. Then use the GUI toolbar to select desired views.

When using the CLI, the command:

```
openss -cli -f mexe-pcsamp.openss
```

loads the database file. Then use the **expview** command options for desired views.

17.4 osscompare: Compare Database Files

General form:

```
osscompare "<db_file1>, <db_file2>[,<db_file>...]" [ time | percent | <other  
metrics> ] [rows=nn] [viewtype=functions| statements | linkedobjects ] > [ oname =  
<csv filename> ]
```

Where:

"<db_file>" represents an O|SS database file created by running an O|SS experiment on an application.

[time | percent | <other metrics>] represent the metric that the comparison will use to differentiate the performance information for each experiment database.

[rows=nn] indicates how many rows of output you want to have listed.

[viewtype=functions| statements | linkedobjects] selects the granularity of the view output. The comparison is either done at the function, statement, or library view level. Function level is the default granularity.

[oname = <csv filename>] Name the output filename when comma separated list output is requested.

Example:

```
osscompare "smg-run1.openss,smg-run2.openss" "f1,f2,f3" "SEP"
```

```
osscompare "smg-run1.openss,smg-run2.openss" percent rows=10
```

Please type "man osscompare" for more details.

17.5 osspcsamp: Program Counter Experiment

General form:

```
osspcsamp "<command> <args>" [ high | low | default | <sampling rate>]
```

Sequential job example:

```
osspcsamp "smg2000 -n 50 50 50"
```

Parallel job example:

```
osspcsamp "mpirun -np 128 smg2000 -n 50 50 50"
```

Additional arguments:

high: twice the default sampling rate (samples per second)

low: half the default sampling rate

default: default sampling rate is 100

<sampling rate>: integer value sampling rate

17.6 ossusertime: Call Path Experiment

General form:

```
ossusertime "<command> <args>" [ high | low | default | <sampling rate>]
```

Sequential job example:

```
ossusertime "smg2000 -n 50 50 50" ["L"]["SEP"]
```

Parallel job example:

```
ossusertime "mpirun -np 64 smg2000 -n 50 50 50"
```

Additional arguments:

high: twice the default sampling rate (samples per second)

low: half the default sampling rate

default: default sampling rate is 35

<sampling rate>: integer value sampling rate

17.7 osshwc, osshwctime: HWC Experiments

General form:

```
osshwc[time] "<command> <args>" [ default | <PAPI_event> | <PAPI threshold> |  
<PAPI_event><PAPI threshold> ]
```

Sequential job example:

```
osshwc[time] "smg2000 -n 50 50 50"[L][SEP]
```

Parallel job example:

```
osshwc[time] "mpirun -np 128 smg2000 -n 50 50 50"[L][SEP]
```

Additional arguments:

default: event (PAPI_TOT_CYC), threshold (10000)

<PAPI_event>: PAPI event name

<PAPI threshold>: PAPI integer threshold

17.8 osshwcsamp: HWC Experiment

General form:

```
osshwcsamp "<command>< args>" [ default | <PAPI_event_list> |  
<sampling_rate> ]
```

Sequential job example:

```
osshwcsamp "smg2000 -n 50 50 50"
```

Parallel job examples:

```
osshwcsamp "mpirun -np 128 smg2000 -n 50 50 50"
```

```
osshwcsamp "srun -N 32 -n 128 sweep3d.mpi" PAPI_L1_DCM,PAPI_L1_DCA 200
```

Additional arguments:

default: events(PAPI_TOT_CYC and PAPI_FP_OPS), sampling_rate is 100

<PAPI_event_list>: Comma separated PAPI event list^[L]_[SEP]

<sampling_rate>: integer value sampling rate

17.9 ossio, ossiot: I/O Experiments

General form:

```
ossio[t] "<command> < args>" [ default | f_t_list ]
```

Sequential job example:

```
ossio[t] "smg2000 -n 50 50 50"
```

Parallel job example:

```
ossio[t] "mpirun -np 128 smg2000 -n 50 50 50"
```

Additional arguments:

default: trace all I/O functions

<f_t_list>: Comma-separated list of I/O functions to trace; one or more of the following: **close**, **creat**, **creat64**, **dup**, **dup2**, **lseek**, **lseek64**, **open**, **open64**, **pipe**, **pread**, **pread64**, **pwrite**, **pwrite64**, **read**, **readv**, **write** and **writew**

17.10 ossmpi, ossmpip, ossmpit: MPI Experiments

General form:

ossmpi[p|t] "<mpirun><mpiargs><command><args>" [**default** | f_t_list]

Parallel job example:

```
ossmpi[p|t] "mpirun -np 128 smg2000 -n 50 50 50"
```

Additional arguments:

default: trace all MPI functions

<f_t_list>: Comma-separated list of MPI functions to trace, consisting of zero or more of: **MPI_Allgather**, **MPI_Waitsome** and/or zero or more of the MPI group categories:

MPI Category	Argument
All MPI Functions	all
Collective Communicators	collective_com
Persistent Communicators	persistent_com
Synchronous Point to Point	synchronous_p2p
Asynchronous Point to Point	asynchronous_p2p
Process Topologies ^[1] _{SEP}	process_topologies
Groups Contexts	graphs_contexts_comms
Communicators	environment ^[1] _{SEP}
Environment ^[1] _{SEP}	datatypes
Datatypes	fileio
MPI File I/O	

17.11 ossmem: Memory Analysis Experiment

General form:

ossmem "<command> <args>" [**default** | f_t_list]

Sequential job example:

```
ossmem "smg2000 -n 50 50 50"
```

Parallel job example:

```
ossmem "mpirun -np 128 smg2000 -n 50 50 50"
```


Additional arguments:

default: trace all supported memory functions

<f_t_list>: Comma-separated list of exceptions to trace, consisting of one or more of: **malloc**, **free**, **memalign**, **posix_memalign**, **calloc** and **realloc**

17.12 ossomptp: OpenMP Specific Profiling Experiment

General form:

ossomptp "<command> <args>"

Sequential job example:

```
ossomptp "openmp_stress < stress.input"
```

Parallel job example:

```
ossomptp "mpirun -np 128 openMP_MD"
```

17.13 osspthreads: POSIX Thread Analysis Experiment

General form:

osspthreads "<command> <args>" [**default** | f_t_list]

Sequential job example:

```
osspthreads "smg2000 -n 50 50 50"
```

Parallel job example:

```
osspthreads "mpirun -np 128 smg2000 -n 50 50 50"
```

Additional arguments:

default: trace all POSIX thread functions

<f_t_list>: Comma-separated list of exceptions to trace, consisting of one or more of: **pthread_create**, **pthread_mutex_init**, **pthread_mutex_destroy**, **pthread_mutex_lock**, **pthread_mutex_trylock**, **pthread_mutex_unlock**, **pthread_cond_init**, **pthread_cond_destroy**, **pthread_cond_signal**, **pthread_cond_broadcast**, **pthread_cond_wait**, and **pthread_cond_timedwait**

17.14 osscuda: NVIDIA CUDA Tracing Experiment

General form:

osscuda "<command> <args>"

Sequential job example:

```
osscuda "eigenvalues --matrix-size=4096"
```

Parallel job example:

```
osscuda "mpirun -np 64 -npernode 1 lmp_linux -sf gpu < in.lj"
```

17.15 cbtfsummary: Overview/Summary Multiple Metric Experiment

General form:

```
cbtfsummary "<command> < args>"
```

Sequential job example:

```
cbtfsummary "./matmul < matmul.input"
```

Parallel job example:

```
osscuda "mpirun -np 128 smg2000 -n 50 50 50"
```

17.16 Key Environment Variables

--	--

Execution Related Variables	Description
OPENSS_RAWDATA_DIR	Used on cluster systems where a /tmp file system is unique on each node. It specifies the location of a shared file system path which is required for O SS to save the “raw” data files on distributed systems. OPENSS_RAWDATA_DIR= “shared file system path” Example: export OPENSS_RAWDATA_DIR=/lustre4/fsys/userid
OPENSS_ENABLE_MPI_PCONTROL	Activates the MPI_Pcontrol function recognition; otherwise O SS will ignore MPI_Pcontrol function calls.
OPENSS_DATABASE_ONLY	When running the O SS convenience scripts, only create the database file and do NOT put out the default report. Used to reduce the size of the batch file output files if user is not interested in looking at the default report.
OPENSS_RAWDATA_ONLY	When running the O SS convenience scripts, only gather the performance information into the OPENSS_RAWDATA_DIR directory, but do NOT create the database file and do NOT put out the default report.
OPENSS_DB_DIR	Specifies the path to where O SS will build the database file. On a file system without file locking enabled, the SQLite component cannot create the database file. This variable is used to specify a path to a file system with locking enabled for the database file creation. This usually occurs on Lustre file systems that don’t have locking enabled. OPENSS_DB_DIR= “file system path” Example: export OPENSS_DB_DIR=/opt/filesys/userid

OPENSS_MPI_IMPLEMENTATION	<p>Specifies the MPI implementation the application is using; only needed for the mpi, mpit, and mpip experiments. These are the currently supported MPI implementations: openmpi, lammpi, mpich, mpich2, mpt, lam, mvapich, mvapich2. For Cray, IBM and Intel MPI implementations, use mpich2.</p> <p>OPENSS_MPI_IMPLEMENTATION="MPI impl. name" Example: export OPENSS_MPI_IMPLEMENTATION=openmpi</p> <p>In most cases, O SS can auto-detect the MPI in use.</p>
OPENSS_DEFER_VIEW	<p>Allow overriding display of the default view for cases where users may not want or need it displayed.</p>
CBTF_CSVDATA_DIR	<p>Sets directory path for the location for the cbtfsummary experiment csv files.</p>

Appendix A: cbtfsuammary csv file format

Application and Host/Rank/Thread Information

DESCRIPTION Of Data (Rows 1-2)	Host that the data was obtained from	Process Id for this data	MPI rank for this data	Thread id for this data	POSIX thread id	Application Name	Total time in Seconds
Row 1: header information	host	pid	rank	tid	posix_tid	executable	total_time_seconds
Row 2: data matching row 1 header	localhost	19879	1	3	140376294303616	lulesh2.0	4.909129

Information in row 1 (header) and 2 (values corresponding to the header):

- Host that the data was obtained from
- Process Id
- MPI rank
- OpenMP thread Id
- POSIX thread Id
- Application name
- Application run time in seconds

Notes on Row 1 - 2:

- The host, process, etc. information is currently always in the first two rows of the csv file

Potential Derived/Non-Derived Information Displayed in Report Form:

- The host, process, etc. information is currently always in the first two rows of the csv file

getrusage type Information

DESCRIPTION Of Data (Rows 3-4)	Maximum resident storage size	User time in seconds	System time in seconds
Row 3: header information	maxrss_bytes	utime_seconds	stime_seconds
Row 4: data matching row 3 header	45644	1.839830	0.539242

Information in row 3 (header) and 4 (values corresponding to the header):

- Maximum resident storage size (high water mark) obtained from getrusage
- User time seconds obtained from getrusage
- System time in seconds obtained from getrusage

Notes on Row 3 - 4:

- The rusage information is currently always in the first third and fourth rows of the csv file

PAPI dmem (Dynamic Memory) Information

DESCRIPTION Of Data (Rows 5-6)	Dynamic memory size	Dynamic resident memory usage	Dynamic memory usage high water mark	Dynamic memory usage that was in shared memory	Dynamic memory usage that was in heap memory
Row 5: header information	dmem_size	dmem_resident	dmem_high_water_mark	dmem_shared	dmem_heap
Row 6: data matching row 1 header	301568	30952	45644	12100	38176

Information in row 5 (header) and 6 (values corresponding to the header):

- Dynamic memory sized obtained from the papi dmem interface
- Dynamic memory resident size obtained from the papi dmem interface
- Dynamic memory high water mark obtained from the papi dmem interface
- Dynamic memory usage that was in shared memory obtained from the papi dmem interface
- Dynamic memory usage that was in heap memory obtained from the papi dmem interface

Notes on Row 5 – 6:

- The papi dmem information is currently always in the first fifth and sixth rows of the csv file

POSIX I/O Information

DESCRIPTION Of Data (I/O if present)	Time spent in POSIX I/O calls	Time spent in POSIX read I/O calls	Time spent in POSIX write I/O calls	Number of bytes via POSIX read I/O calls	Number of bytes via POSIX write I/O calls
I/O Row 1: header information	io_total_time	read_time	write_time	read_bytes	writebytes

I/O Row 2: data matching row 1 header	0.000120	0.000059	0.000061	13618	1961
---------------------------------------	----------	----------	----------	-------	------

Information in I/O row 1 (header) and I/O row 2 (values corresponding to the header):

- Time spent in POSIX I/O calls obtained from cbtf I/O wrappers
- Time spent in POSIX read I/O calls obtained from cbtf I/O wrappers
- Time spent in POSIX write I/O calls obtained from cbtf I/O wrappers
- Number of bytes read via POSIX read I/O calls obtained from cbtf I/O wrappers
- Number of bytes written via POSIX write I/O calls obtained from cbtf I/O wrappers

Notes on I/O Rows:

- If I/O is present this information will (currently) show up in row 7 and 8 of the csv file

POSIX Memory Allocation Call Information

DESCRIPTION Of Data (mem rows)	Time spent in allocation (malloc, calloc, realloc, etc) calls	Number of allocation calls	Number of bytes allocated
Mem Row 1: header information	allocation_time	allocation_calls	allocation_bytes
Mem Row 2: data matching row 1 header	0.000001	5	317

Information in mem row 1 (header) and mem row 2 (values corresponding to the header):

- Time spent in allocation (malloc, calloc, realloc, etc) calls obtained from cbtf mem wrappers
- Number of allocation (malloc, calloc, realloc, etc) calls obtained from cbtf mem wrappers
- Number of bytes allocated via allocation (malloc, calloc, realloc, etc) calls obtained from cbtf mem wrappers

Notes on Row mem rows:

- TBD

POSIX Memory Free Call Information

DESCRIPTION Of Data (Mem free rows)	Time spent inf free calls	Number of calls to free
Mem free row 1: header information	free_time	free_calls
Mem free row 2: data matching row 1 header	0.000001	1

Information in mem free row 1 (header) and mem free row 2 (values corresponding to the header):

- Time spent in memory free calls obtained from cbtf mem wrappers
- Number of free calls obtained from cbtf mem wrappers

Notes on mem free rows:

- TBD

MPI Function Call Information

DESCRIPTION Of Data (MPI rows)	Total time spent in MPI functions
MPI Row 1: header information	total_mpi_time
MPI Row 2: data matching row 1 header	0.340121

Information in MPI row 1 (header) and MPI row 2 (values corresponding to the header):

- Time spent in MPI function calls obtained from cbtf mpi wrappers

Notes on MPI rows:

- This could be refined in the future to contain timing by MPI categories.

Hardware Counter Event Count Information

DESCRIPTION Of Data (HWC Rows)	PAPI counter for total cycles	PAPI counter for total instructions	PAPI counter for load instructions	PAPI counter for level 3 total cache misses	PAPI counter for level 2 total cache misses	PAPI counter for level 1 total cache misses	More papi counters.....
HWC row 1: header information	PAPI_TOT_CYC	PAPI_TOT_INS	PAPI_LD_INS	PAPI_L3_TCM	PAPI_L2_TCM	PAPI_L1_TCM	...
HWC row 2: data matching row 1 header	6100004039	5511171256	1950917099	9663853	20989452	77687293	...

Information in HWC row 1 (header) and HWC row 2 (values corresponding to the header):

- Hardware counter event occurrences obtained from cbtf hardware counter collectors

- List of default hardware counters that are cycled through and used if they are found to be available on the system the application is running on:
 - PAPI_TOT_CYC Total cycles
 - PAPI_TOT_INS Total instructions
 - PAPI_LD_INS Load instructions
 - PAPI_VEC_DP Double precision vector/SIMD instructions
 - PAPI_DP_OPS Double precision floating point operations
 - PAPI_FDV_OPS Floating point divide operations
 - PAPI_FP_INS Floating point instructions
 - PAPI_FP_OPS Floating point operations
 - PAPI_L3_TCM Level 3 cache misses
 - PAPI_L2_TCM Level 2 cache misses
 - PAPI_L1_TCM Level 1 cache misses
 - PAPI_TLB_IM Instruction translation lookaside buffer misses
 - PAPI_REF_CYC Reference clock cycles
 - PAPI_REF_NS
 - PAPI_FUL_CCY Cycles with maximum instructions completed
 - PAPI_RES_STL Cycles stalled on any resource

Notes on HWC rows:

- This could be refined in the future to contain timing by MPI categories.

OpenMP Time Information

DESCRIPTION Of Data (OpenMP Rows)	Implicit Task Time	Serial time	Barrier time	Barrier wait time	Idle time	
OpenMP Row 1: header information	implicit_task_time	serial_time	barrier_time	wait_barrier_time	idle_time	
OpenMP Row 2: data matching row 1 header	4.903256	0.006370	0.0	0.0	18446694400.000000	

Information in OpenMP row 1 (header) and OpenMP row 2 (values corresponding to the header):

- Implicit task time obtained from cbtf omptp based OpenMP wrappers
- Serial task time obtained from cbtf omptp based OpenMP wrappers
- Barrier task time obtained from cbtf omptp based OpenMP wrappers
- Wait Barrier task time obtained from cbtf omptp based OpenMP wrappers
- Idle task time obtained from cbtf omptp based OpenMP wrappers

Notes on OpenMP rows:

- TBD