

## 9 Next Generation O|SS GUI Application

### 9.1 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 Open|SpeedShop sampling and tracing experiments such as “pcssamp”, “usertime”, “hwc”, “hwctime”, “hwcsamp”, “omptp”, “mem”, “io”, “iop”, “iot”, “mpi”, “mpip”, “mpit” and “pthreads”.

The Next Generation Open|SpeedShop 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 Open|SpeedShop 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.

### 9.1.1 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

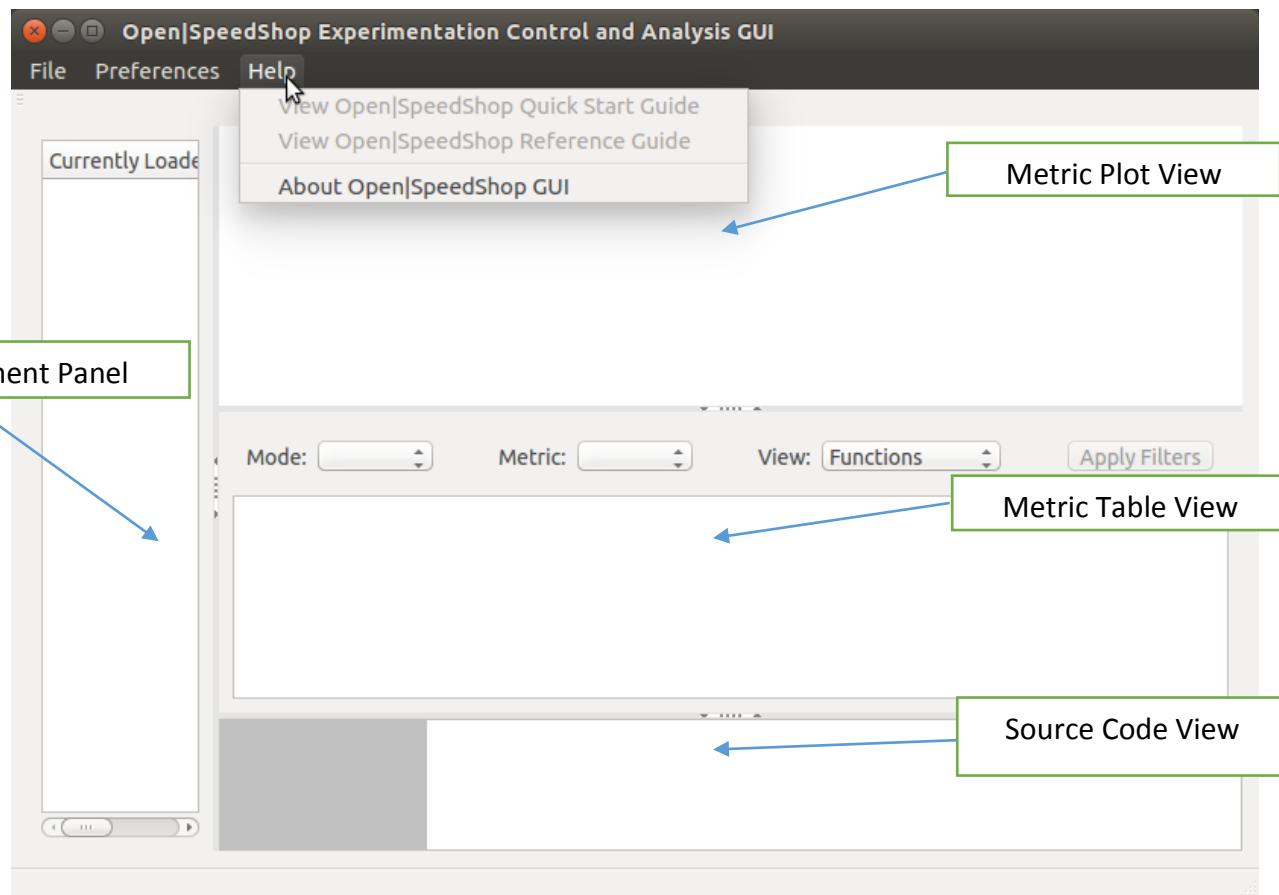


Figure 1 - Main Window User Interface Layout

The main window has a menu bar with three menu items - “File”, “Preferences” and “Help”.

Figure 1 shows the “Help” menu items which are:

- View Open|SpeedShop Quick Start Guide
- View Open|SpeedShop Reference Guide
- About Open|SpeedShop GUI

If the Open|SpeedShop 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 Open|SpeedShop 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 Open|SpeedShop website home page.



Figure 2 - About Open|SpeedShop GUI dialog

Figure 3a shows the “File” menu items “Load O|SS Experiment” and “Unload O|SS Experiment”.

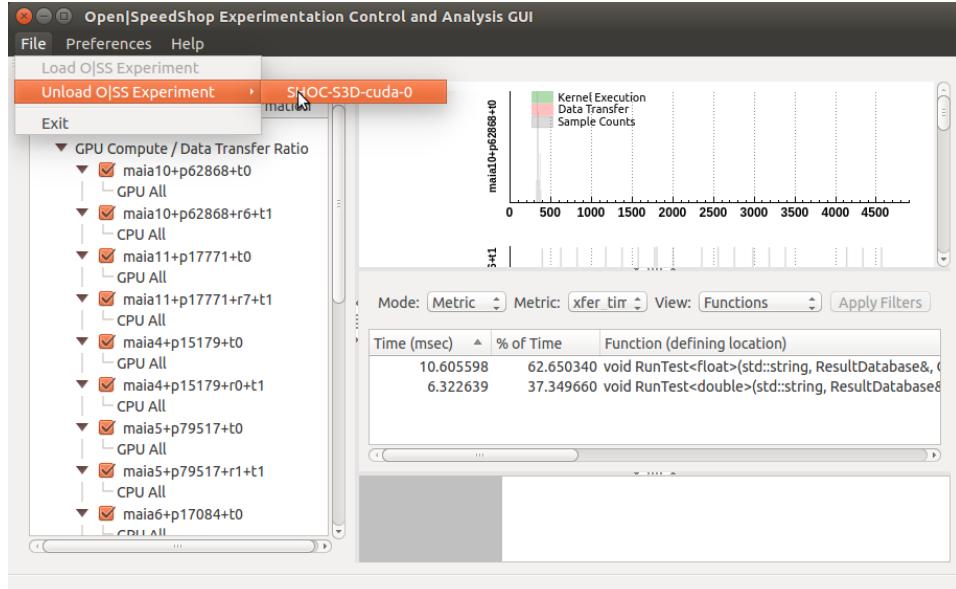


Figure 3a - File menu

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 3a). 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 3b).

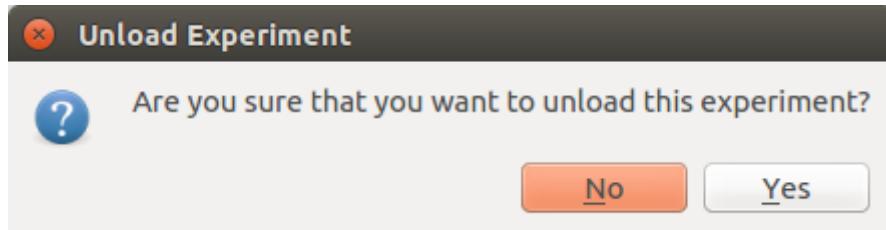


Figure 3b - Confirm Unload Experiment Dialog

Figure 4a shows the “Preferences” menu item “Show Derived Metrics” and Figure 4b shows the “Derived Metrics Information” dialog that is shown upon activating the “Show Derived Metrics” menu item.

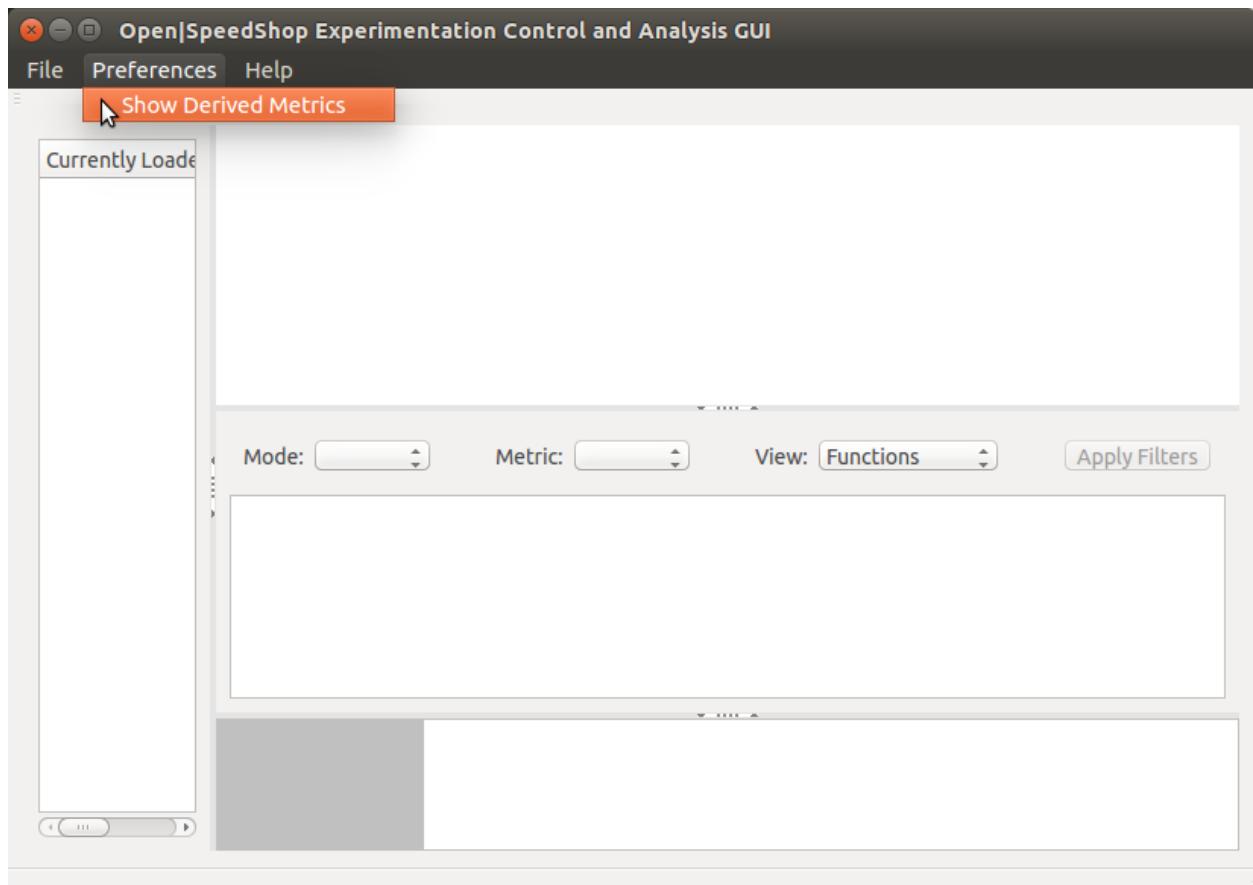


Figure 4a - Preferences Menu

**Derived Metrics Information**

**Derived Metric Definitions**

Name / Description	Formula	Enabled
% of Cycles Stalled on Any Resource	$100.0 * (\text{PAPI\_RES\_STL} / \text{PAPI\_TOT\_CYC})$	<input checked="" type="checkbox"/>
% of Cycles Stalled on Memory Access	$100.0 * (\text{PAPI\_MEM\_SCY} / \text{PAPI\_TOT\_CYC})$	<input checked="" type="checkbox"/>
% of Cycles Waiting for Memory Access	$100.0 * (\text{PAPI\_STL\_SCY} / \text{PAPI\_TOT\_CYC})$	<input checked="" type="checkbox"/>
% of Cycles with no instruction issue	$100.0 * (\text{PAPI\_STL\_ICY} / \text{PAPI\_TOT\_CYC})$	<input checked="" type="checkbox"/>
Data References Per Instruction	$\text{PAPI\_L1\_DCA} / \text{PAPI\_TOT\_INS}$	<input checked="" type="checkbox"/>
FP Instructions Per Cycle	$\text{PAPI\_FP\_INS} / \text{PAPI\_TOT\_CYC}$	<input checked="" type="checkbox"/>
Graduated Instructions / Issued Instructions	$\text{PAPI\_FP\_INS} / \text{PAPI\_TOT\_IIS}$	<input checked="" type="checkbox"/>
Instructions Per Cycle	$\text{PAPI\_TOT\_INS} / \text{PAPI\_TOT\_CYC}$	<input checked="" type="checkbox"/>
Issued Instructions Per Cycle	$\text{PAPI\_TOT\_IIS} / \text{PAPI\_TOT\_CYC}$	<input checked="" type="checkbox"/>
L1 Cache Data Hit Rate	$1.0 - (\text{PAPI\_L1\_DCM} / \text{PAPI\_LST\_INS})$	<input checked="" type="checkbox"/>
L1 Cache Line Reuse (data)	$(\text{PAPI\_LST\_INS} - \text{PAPI\_L1\_DCM}) / \text{PAPI\_L1\_DCM}$	<input checked="" type="checkbox"/>
L1 Data Cache Read Miss Ratio	$\text{PAPI\_L1\_DCM} / \text{PAPI\_L1\_DCA}$	<input checked="" type="checkbox"/>
L2 Cache Data Hit Rate	$1.0 - (\text{PAPI\_L2\_DCM} / \text{PAPI\_L1\_DCM})$	<input checked="" type="checkbox"/>
L2 Cache Line Reuse (data)	$(\text{PAPI\_L1\_DCM} - \text{PAPI\_L2\_DCM}) / \text{PAPI\_L2\_DCM}$	<input checked="" type="checkbox"/>
L2 Cache Miss Ratio	$\text{PAPI\_L2\_TCM} / \text{PAPI\_L2\_TCA}$	<input checked="" type="checkbox"/>
L3 Cache Data Hit Rate	$1.0 - (\text{PAPI\_L3\_DCM} / \text{PAPI\_L2\_DCM})$	<input checked="" type="checkbox"/>
L3 Cache Data Read Ratio	$\text{PAPI\_L3\_DCR} / \text{PAPI\_L3\_DCA}$	<input checked="" type="checkbox"/>
L3 Cache Instruction Miss Ratio	$\text{PAPI\_L3\_ICM} / \text{PAPI\_L3\_ICR}$	<input checked="" type="checkbox"/>
L3 Cache Line Reuse (data)	$(\text{PAPI\_L2\_DCM} - \text{PAPI\_L3\_DCM}) / \text{PAPI\_L3\_DCM}$	<input checked="" type="checkbox"/>
L3 Data Cache Miss Ratio	$\text{PAPI\_L3\_DCM} / \text{PAPI\_L3\_DCA}$	<input checked="" type="checkbox"/>
L3 Total Cache Miss Ratio	$\text{PAPI\_L3\_TCM} / \text{PAPI\_L3\_TCA}$	<input checked="" type="checkbox"/>
Percentage FP Instructions	$\text{PAPI\_FP\_INS} / \text{PAPI\_TOT\_INS}$	<input checked="" type="checkbox"/>
Ratio L1 Data Cache Miss to Total Cache Access	$\text{PAPI\_L1\_DCM} / \text{PAPI\_L1\_TCA}$	<input checked="" type="checkbox"/>
Ratio L2 Data Cache Miss to Total Cache Access	$\text{PAPI\_L2\_DCM} / \text{PAPI\_L2\_TCA}$	<input checked="" type="checkbox"/>
Ratio L3 Total Cache Miss to Data Cache Access	$\text{PAPI\_L3\_TCM} / \text{PAPI\_L3\_DCA}$	<input checked="" type="checkbox"/>
Ratio Mispredicted to Correctly Predicted Branches	$\text{PAPI\_BR\_MSP} / \text{PAPI\_BR\_PRC}$	<input checked="" type="checkbox"/>

Figure 4b - Derived Metrics Information 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 experiment database (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 Open|SpeedShop, 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.

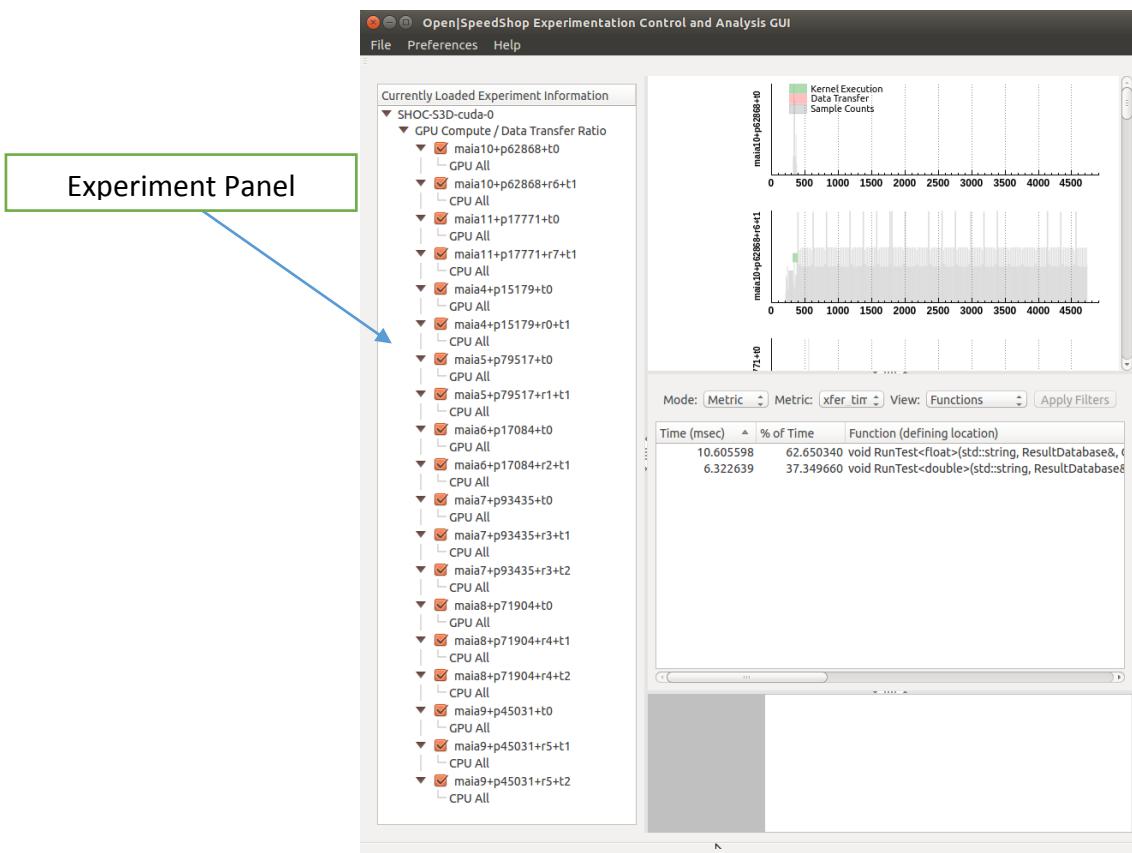


Figure 5 - Experiment Panel

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 showing the detailed performance information collected and computed metric data. 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 O|SS 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”, “Derived 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.

For the “Derived Metric” view mode, currently only selectable when an “hwcsamp” experiment has been loaded, will show derived metric parameters computed in accordance with the formulas and enabled state shown in the Derived Metrics Information dialog (see Figure 4b).

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 term 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).

```

$ 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 descending order sorting and the downward triangle icon is ascending 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, 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, 397ul, 31ul, 2567483615ul, 11ul, 4294967295ul, 7ul, 2>
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, 11ul, 4294967295ul, 7ul, 2>>
104.300000	1.335519	__round (/build/eglibc-SvCtMH/eglibc-2.19/math/../sysdeps/ieee754/dbl-64/wordsizes-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, 2>

Figure 7 - Column sorting via mouse clicks on column header (descending 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, 2>
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/bits/random.h, 1942)
23.100000	0.295780	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/bits/random.h, 1946)

Figure 8 - Column sorting via mouse clicks on column header (ascending 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). 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.

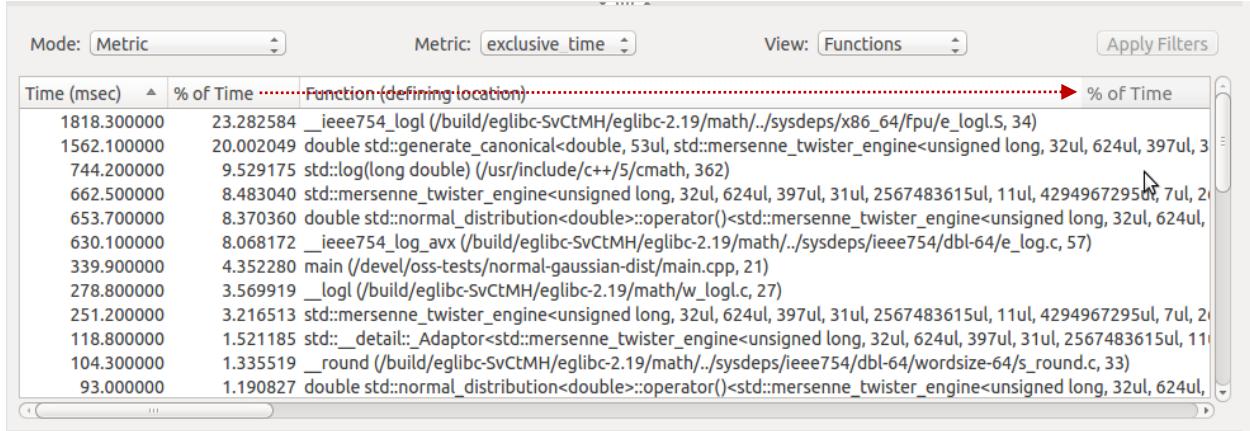


Figure 9 - Changing Column Ordering

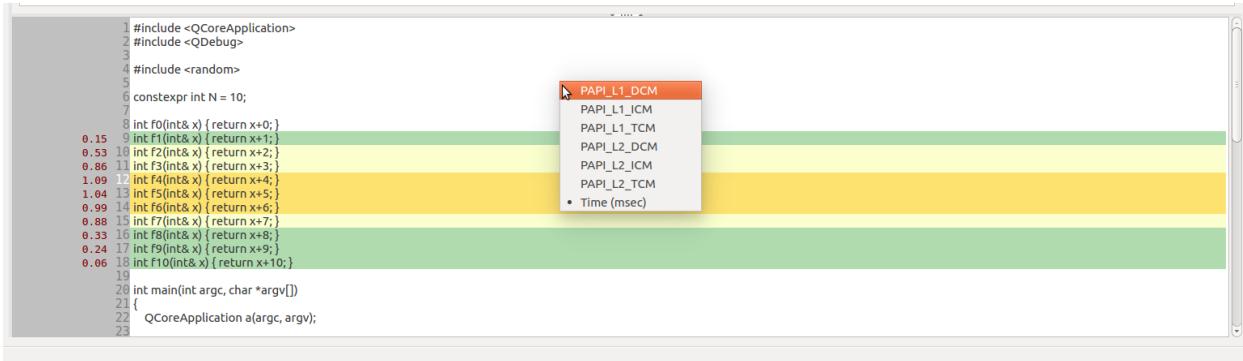
The lower section on the right-hand side is the Source Code View (ref. Figure 10). When the user has activated the “Metric” mode of the Metric Table View, any selections of a row in the table under the “Function (defining location)” column cause the corresponding line of the source code in the Source 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.



Figure 10 - Source Code View (hwcsamp default)

As seen in Figure 10 the left-hand side of the Source Code View has an area to annotate each line of code that has metric values (in red). The line number is displayed in a darker shade of gray then the background gray color of the left-hand side area. The currently selected line number is highlighted in white. Initially the default metric is used to annotate the source-code view. However, depending on the experiment type, there may be other metric events to

choose from. There is a context menu that is activated by holding the right-mouse button (ref. Figure 11). Once the context menu is activated a list of available metric events to choose from is presented. The currently selected metric event has a dot icon in front of it. Once the user has selected a new metric event, the source-code view metric annotations will be updated to show the values for the newly selected metric event (ref. Figure 12).



A screenshot of a source code editor window. A context menu is open over the 15th line of code, which contains the function f1. The menu items are: PAPI\_L1\_DCM, PAPI\_L1\_ICM, PAPI\_L1\_TCM, PAPI\_L2\_DCM, PAPI\_L2\_ICM, PAPI\_L2\_TCM, and Time (msec). The line of code is highlighted in green.

```

1 #include <QCoreApplication>
2 #include <QDebug>
3
4 #include <random>
5
6 constexpr int N = 10;
7
8 int f0(int& x) { return x+0; }
0.15 9 int f1(int& x) { return x+1; }
0.53 10 int f2(int& x) { return x+2; }
0.86 11 int f3(int& x) { return x+3; }
1.09 12 int f4(int& x) { return x+4; }
1.04 13 int f5(int& x) { return x+5; }
0.99 14 int f6(int& x) { return x+6; }
0.88 15 int f7(int& x) { return x+7; }
0.33 16 int f8(int& x) { return x+8; }
0.24 17 int f9(int& x) { return x+9; }
0.06 18 int f10(int& x) { return x+10; }
19
20 int main(int argc, char *argv[])
21 {
22     QCoreApplication a(argc, argv);
23 }
```

Figure 11 - Source Code View (metric event selection context-menu)



A screenshot of a source code editor window. The same context menu is open over the 15th line of code, but now the menu item PAPI\_L1\_DCM is highlighted with a dot icon. The line of code is highlighted in green.

```

1 #include <QCoreApplication>
2 #include <QDebug>
3
4 #include <random>
5
6 constexpr int N = 10;
7
8 int f0(int& x) { return x+0; }
163633 9 int f1(int& x) { return x+1; }
396829 10 int f2(int& x) { return x+2; }
520188 11 int f3(int& x) { return x+3; }
507876 12 int f4(int& x) { return x+4; }
681250 13 int f5(int& x) { return x+5; }
516434 14 int f6(int& x) { return x+6; }
555227 15 int f7(int& x) { return x+7; }
174570 16 int f8(int& x) { return x+8; }
159881 17 int f9(int& x) { return x+9; }
41130 18 int f10(int& x) { return x+10; }
19
20 int main(int argc, char *argv[])
21 {
22     QCoreApplication a(argc, argv);
23 }
```

Figure 12 - Source Code View (metric annotations updated)

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 13).

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.

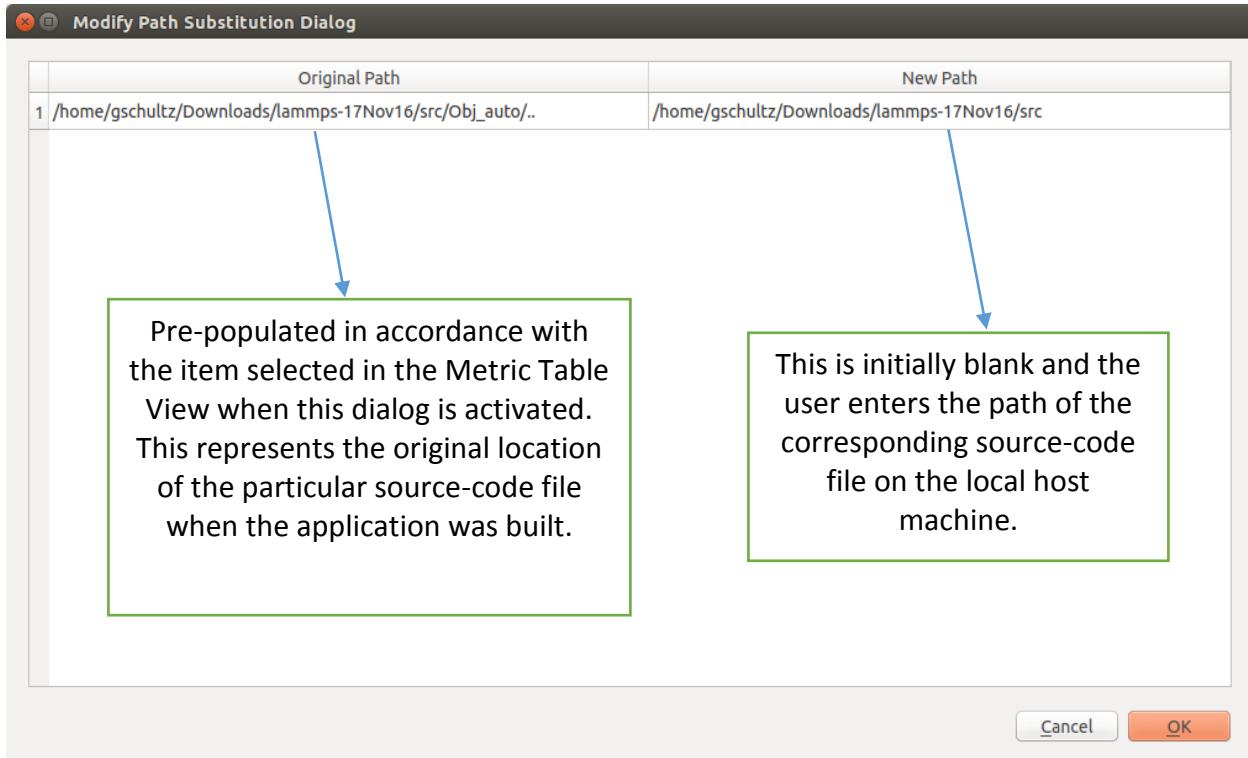


Figure 13 - Modify Path Substitutions Dialog

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 14). 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 15). Once the user has completed the selection of a file in the “Select Directory For File” dialog, the directory path of the selected file is added to the “New Path” entry (ref. Figure 16). The text in the “Original Path” area can be changed by the user as well if needed.

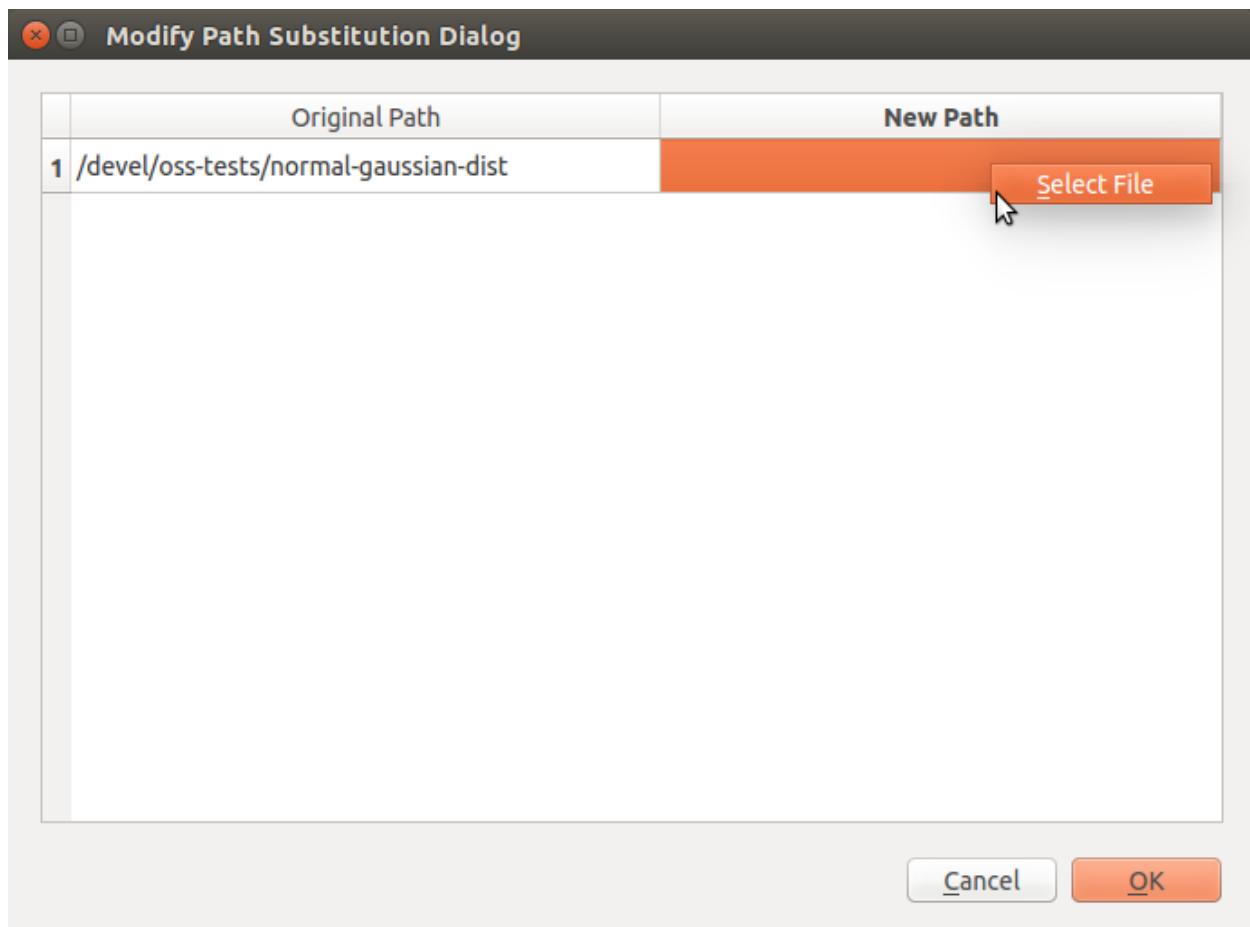


Figure 14 - Modify Path Substitutions Dialog – Select File

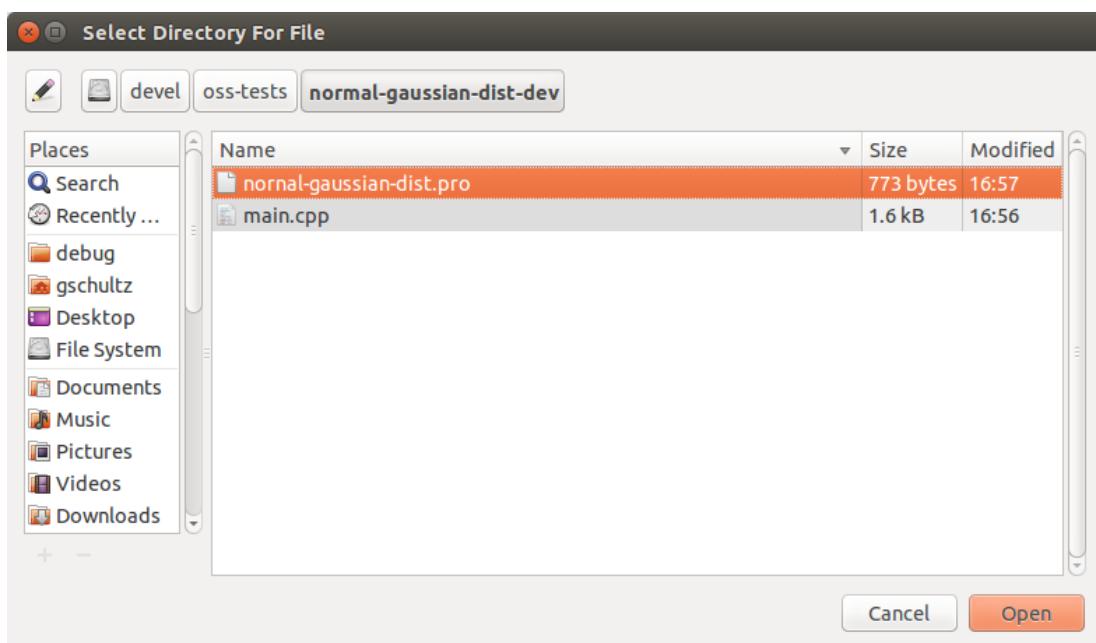


Figure 15 - Select Directory For File Dialog

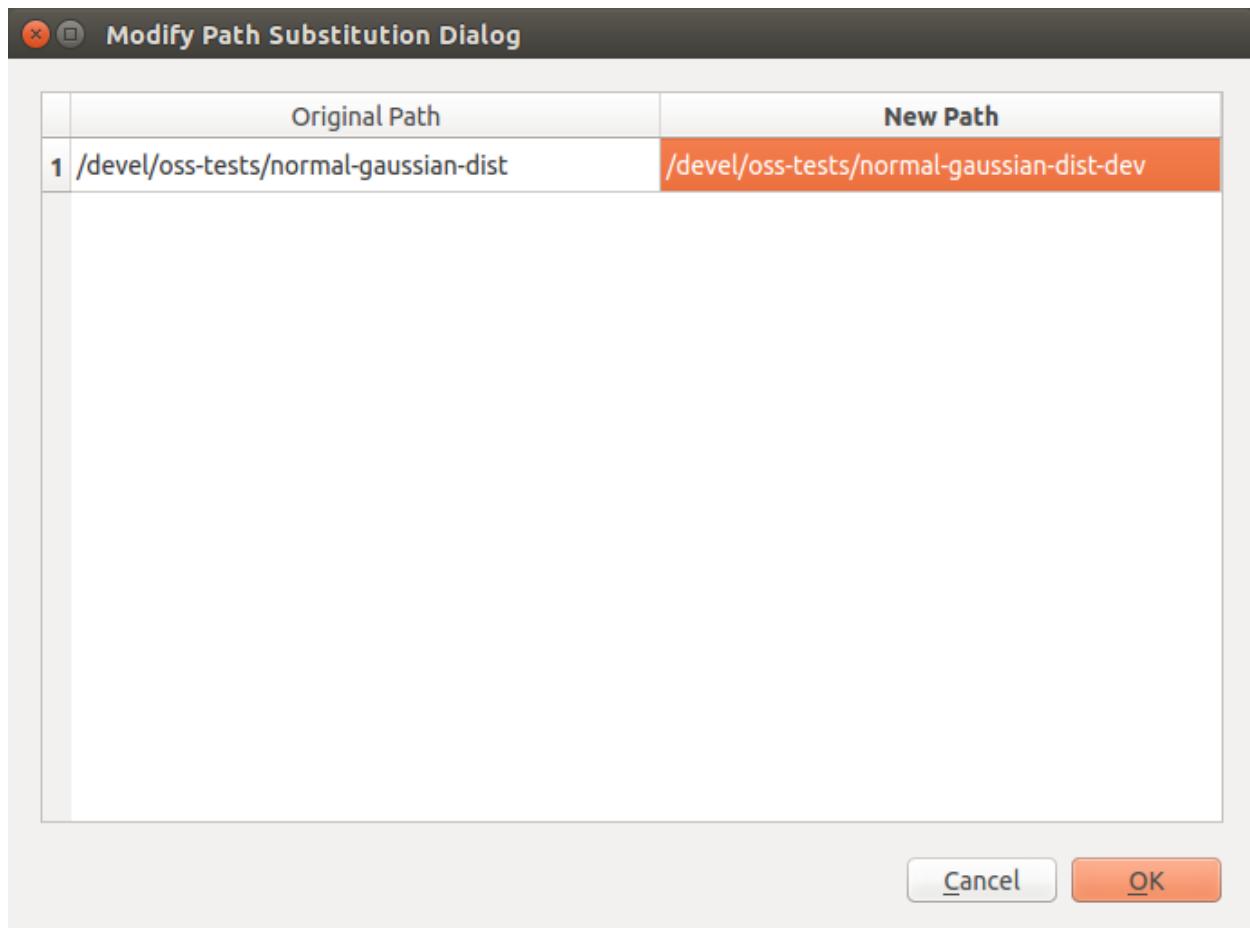


Figure 16 - Completed New Path Entry

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 17).

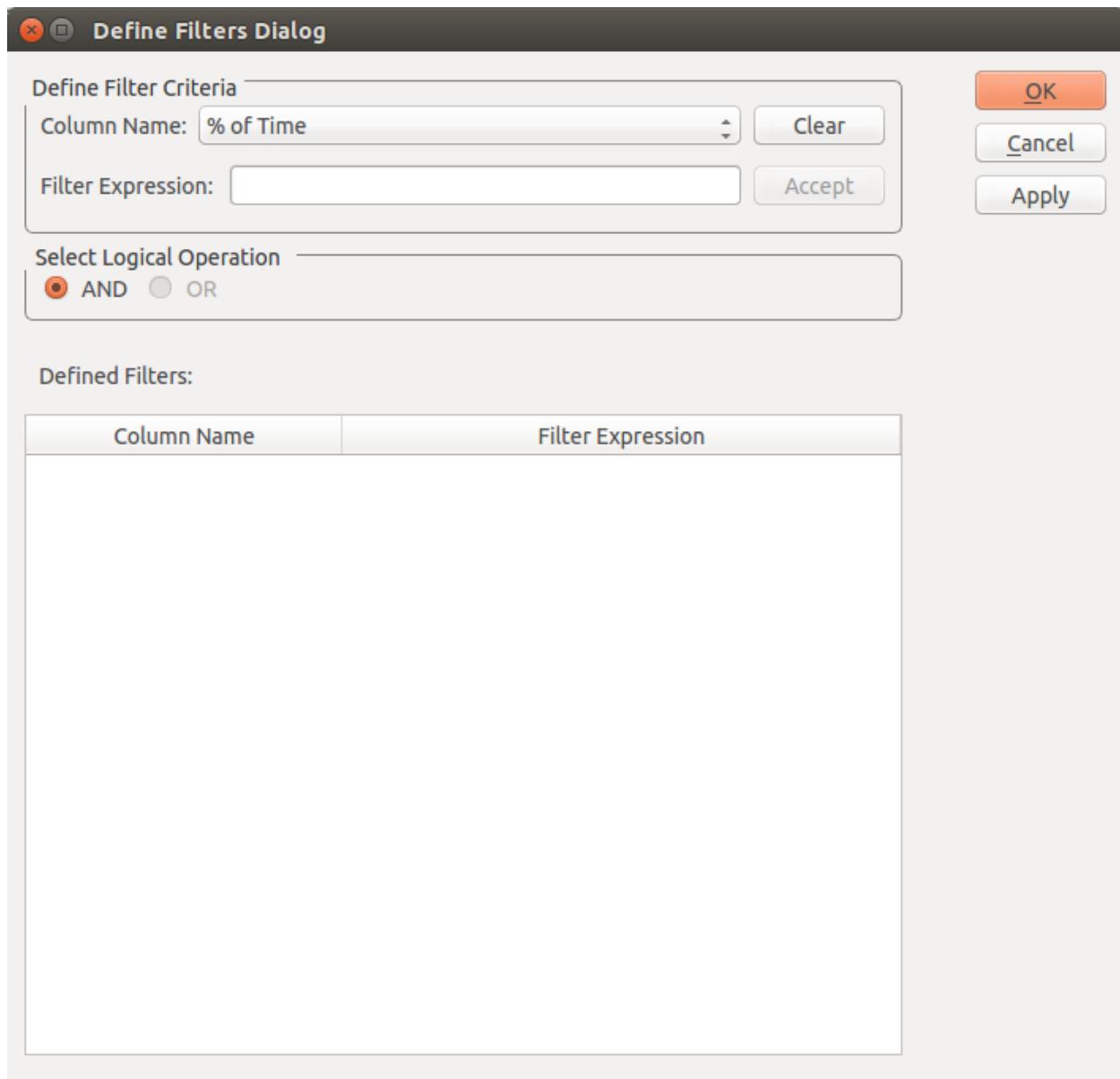


Figure 17 - Define View Filters Dialog

Once the “Define View Filters” dialog appears, the “Column Name” combo-box is used to select a column in the Metric Table View to be used in the filter expression to be defined. The filter expression to be applied to the selected column is entered in the “Filter Expression” text entry area. The filter expression must be any valid regular expression (ref.

[https://en.wikipedia.org/wiki/Regular\\_expression](https://en.wikipedia.org/wiki/Regular_expression)). As entry occurs the filter expression is validated continuously to determine whether it is a valid regular expression. At any time it is a valid regular expression the “Accept” button becomes enabled. Once the user enters a valid filter expression, pressing the “Accept” button will place the new filter in the list of filters for the Metric Table View. Currently the overall filter expression applied is the AND of each

individual filter expression listed. Only those rows returning a true value for the filter expression are displayed. The “Apply” button is used to immediately apply the filter expression to the Metric Table View causing it to update. The “Ok” button will close the dialog but not apply the new filter expression. The filter expression can be applied from the Metric Table View panel by clicking on the “Apply Filters” button (ref. Figure 18). Once filtering is activated the “Apply Filters” button changes to “Clear Filters” (ref. Figure 19). The “Clear Filters” button can be pressed to deactivate filtering in the Metric Table View.

Mode:	Metric	Metric:	exclusive_detail	View:	Functions	
Time (msec)	PAPI_L1_DCM	PAPI_L1_ICM	PAPI_L2_DCM	PAPI_L2_ICM	PAPI_L1_TCM	PAPI_L2_TCM
Function (defining location)						
66.880000	36515458	8510371	28212186	4548426	45025829	32760612 _ieee754_log (/build/eglibc-SvCtMH/eglibc-2.19/math/_/sysdeps/x86_64/fpu/e_logl.S, 34)
59.530000	34477231	7821033	26183030	4156914	42298264	30339944 double std::generate_canonical<double, 53ul, std::mersenne_twister_engine>::operator<<(std::ostream&, double) (/usr/include/c++/5/cmath, 362)
28.360000	16129835	3613966	12079433	1937324	19743801	14016757 std::log(double) (/usr/include/c++/5/cmath, 362)
26.510000	15261292	3493393	11605602	1846648	18754685	1345250 std::mersenne_twister_engine::unsigned long, 32ul, 624ul, 397ul, 31ul, 2567483615ul, 11ul, ...
23.390000	13863485	3131324	10368829	1623784	16994809	11992613 _ieee754_log_avx (/build/eglibc-SvCtMH/eglibc-2.19/math/_/sysdeps/ieee754/dbl-64/e_logl.c, 27)
23.200000	12780154	2833169	9449760	1497332	15613323	10947092 double std::normal_distribution<double>::operator<<(std::ostream&, double) (/usr/include/c++/5/cmath, 362)
12.940000	7148120	1637700	5336619	867877	8785820	6204499 main (/dev/oss-tests/normal-gaussian-dist/main.cpp, 21)
10.310000	5756532	1324152	4463173	713775	7080684	5176948 _log (/build/eglibc-SvCtMH/eglibc-2.19/math/w_logl.c, 27)
10.050000	6354683	1385224	4760963	742803	7739907	5503768 std::mersenne_twister_engine::unsigned long, 32ul, 624ul, 397ul, 31ul, 2567483615ul, 11ul, ...
4.250000	2506973	563069	1924885	299777	3070042	2224662 std::__detail::Adaptor<std::mersenne_twister_engine>::unsigned long, 32ul, 624ul, 397ul, 31ul, 2567483615ul, 11ul, ...
4.110000	2223606	500069	1675082	273571	2723675	1948653 __round (/build/eglibc-SvCtMH/eglibc-2.19/math/_/sysdeps/ieee754/dbl-64/wordsizes-64/sqrt.c, 27)
3.160000	1758289	401978	1342955	218501	2160267	1561456 __sqrt (/build/eglibc-SvCtMH/eglibc-2.19/math/w_sqrt.c, 27)
2.990000	1908752	445741	1507980	230912	2354493	1738892 double std::normal_distribution<double>::operator<<(std::ostream&, double) (/usr/include/c++/5/cmath, 362)
2.880000	1606241	387009	1235487	195077	1993250	1430564 std::__detail::Adaptor<std::mersenne_twister_engine>::unsigned long, 32ul, 624ul, 397ul, 31ul, 2567483615ul, 11ul, ...
2.680000	1526252	352837	1212173	187895	1879362	1400068 std::mersenne_twister_engine::unsigned long, 32ul, 624ul, 397ul, 31ul, 2567483615ul, 11ul, ...
2.150000	1169039	285245	933500	148151	1454284	1081651 unsigned long const& std::min<unsigned long>::operator<<(std::ostream&, unsigned long const&)
1.650000	938545	224184	747932	120151	1162729	868084 unsigned long const& std::max<unsigned long>::operator<<(std::ostream&, unsigned long const&)

Figure 18 - Apply Filters Button

Mode:	Metric	Metric:	exclusive_detail	View:	Functions	
Time (msec)	PAPI_L1_DCM	PAPI_L1_ICM	PAPI_L2_DCM	PAPI_L2_ICM	PAPI_L1_TCM	PAPI_L2_TCM
Function (defining location)						
12.940000	7148120	1637700	5336619	867877	8785820	6204499 main (/dev/oss-tests/normal-gaussian-dist/main.cpp, 21)
1.090000	507876	120240	378819	61576	628116	440395 f4(nt8) (/dev/oss-tests/normal-gaussian-dist/main.cpp, 12)
1.040000	681250	160376	572154	89864	841626	662018 f5(nt8) (/dev/oss-tests/normal-gaussian-dist/main.cpp, 13)
0.990000	516434	117312	400343	63915	633746	464258 f6(nt8) (/dev/oss-tests/normal-gaussian-dist/main.cpp, 14)
0.880000	555227	119316	412997	66955	674543	479952 f7(nt8) (/dev/oss-tests/normal-gaussian-dist/main.cpp, 15)
0.860000	520188	113573	380499	60201	633761	440700 f3(nt8) (/dev/oss-tests/normal-gaussian-dist/main.cpp, 11)
0.530000	396829	81963	274721	38572	478792	313293 f2(nt8) (/dev/oss-tests/normal-gaussian-dist/main.cpp, 10)
0.330000	174570	36007	122436	19256	210577	141692 f8(nt8) (/dev/oss-tests/normal-gaussian-dist/main.cpp, 16)
0.240000	159881	32941	136641	20270	192822	156911 f9(nt8) (/dev/oss-tests/normal-gaussian-dist/main.cpp, 17)
0.150000	163633	40992	154330	23053	204625	177383 f1(nt8) (/dev/oss-tests/normal-gaussian-dist/main.cpp, 9)
0.060000	41130	7323	26815	4441	48453	31256 f10(nt8) (/dev/oss-tests/normal-gaussian-dist/main.cpp, 18)

Figure 19 - Clear Filters Button

## 9.1.4 Case Studies of Using the O|SS GUI to Analyze Experiment Results

The following sections of the User Guide do not necessarily have to be read unless more overall understanding of the GUI is desired. At any time a particular experiment type is going to be used, the user may return to the appropriate section and read or re-read as needed to get background information on what capabilities are provided as well as an example provided in the case study. If there are particular experiment types of interest, then those sections should be read first.

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.

### 9.1.4.1 Using the O|SS GUI to Analyze “pcsample” Experiment Results

Upon loading the “pcsample” experiment the default view appears showing a bar graph of the time distribution across the application functions attributable to the recorded Program Counter (PC) values. These values are also shown in the Metric Table View below the bar graph (ref. Figure 20).

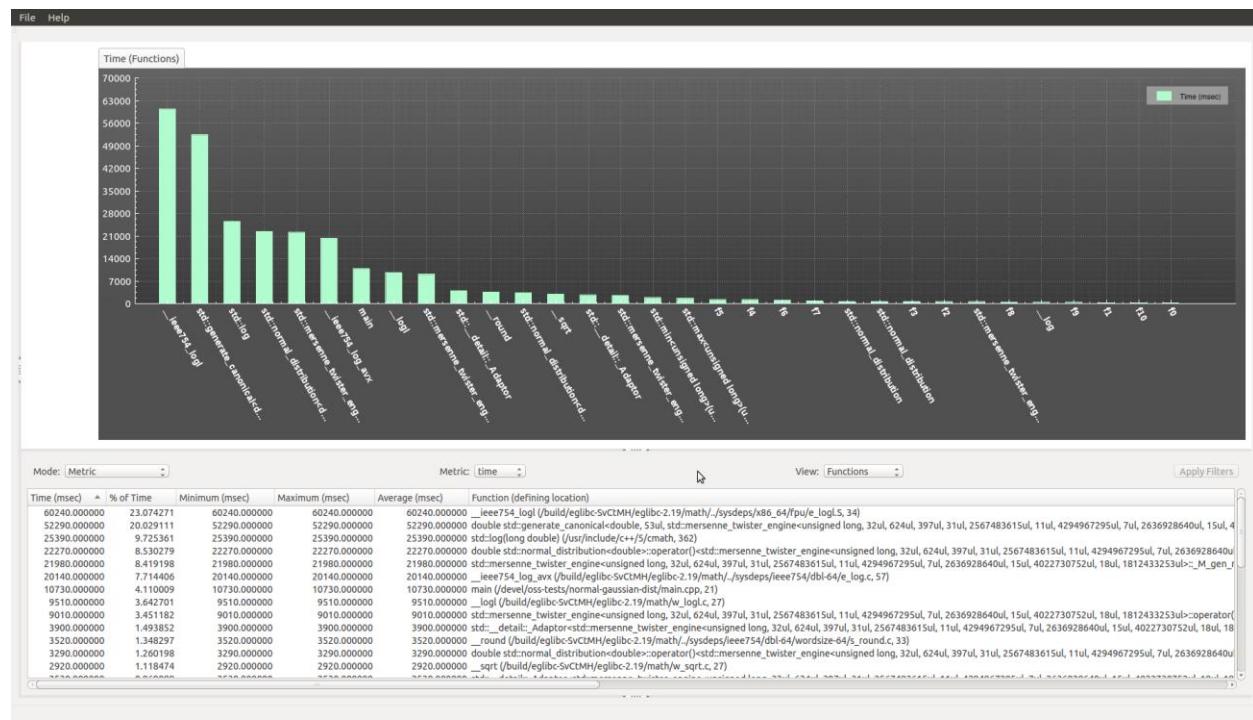


Figure 20 - pc sample experiment default view

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 22).

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 23). 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 24).

Let’s examine the source-code associated with the experiment being discussed in this example (ref. Figure 21). Lines 25 and 26 initialize the random number generator based on the Mersenne Twister algorithm. Line 32 creates an instance of the std::normal\_distribution templated 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 produced 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<10000000000; ++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 21 - 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).

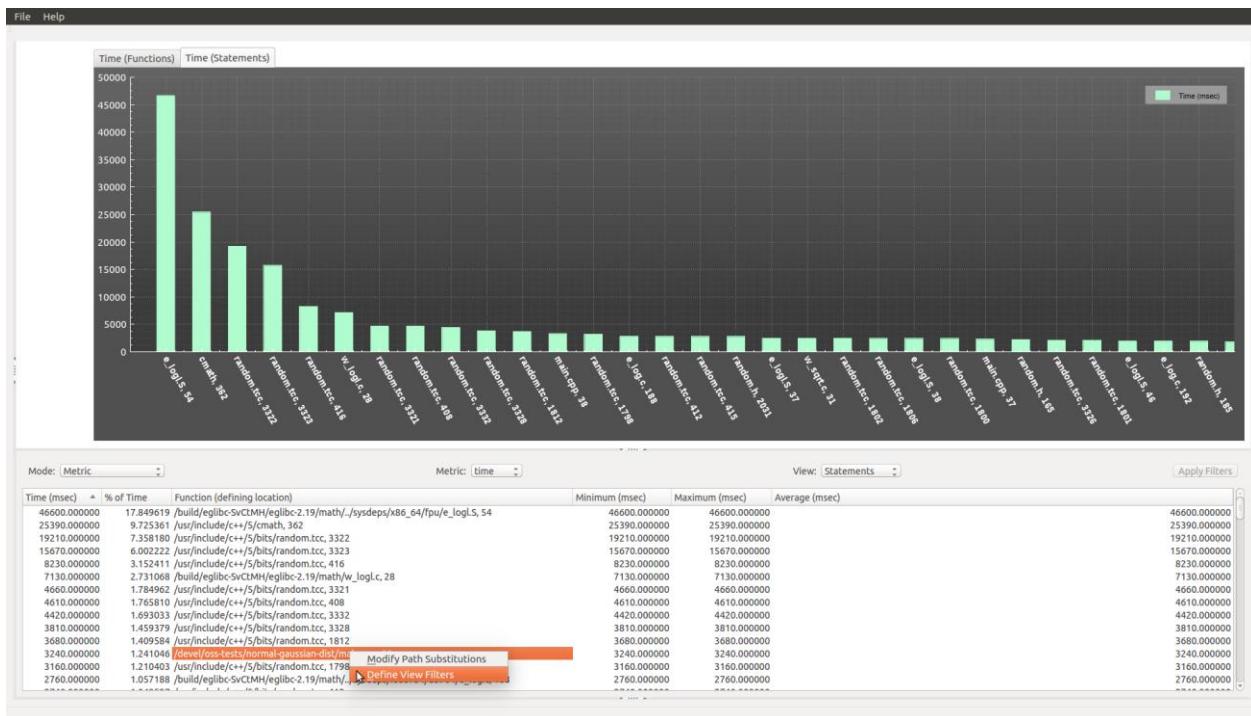


Figure 22 - Activate Define View Filters Dialog

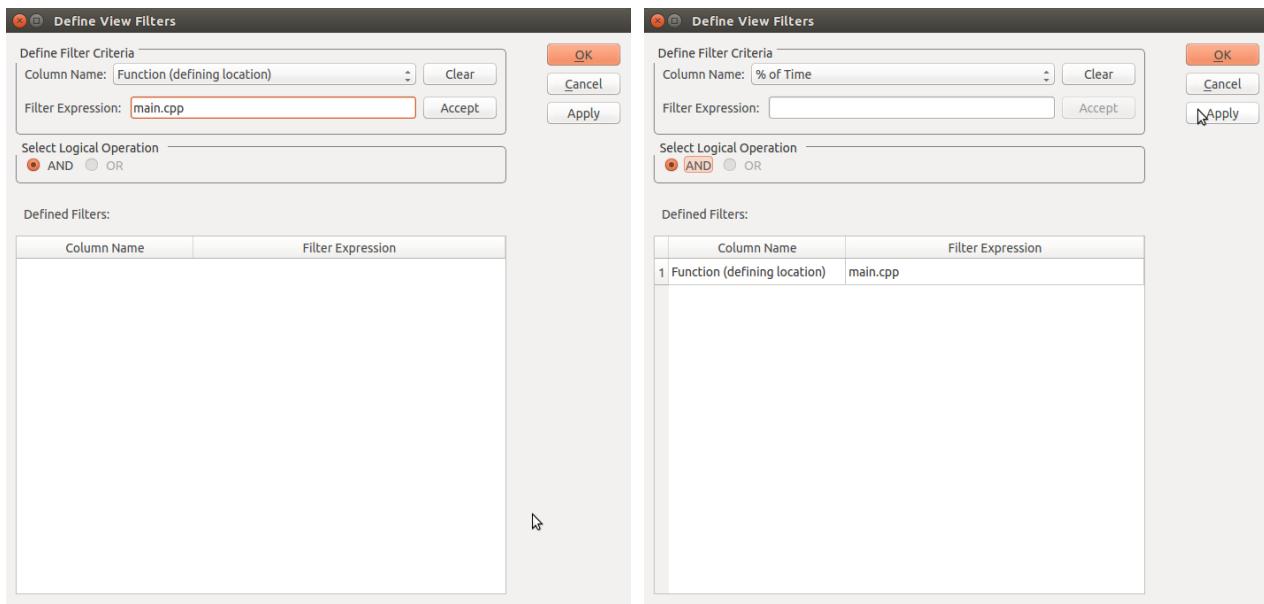


Figure 23 - Define Filter for Metric Table View

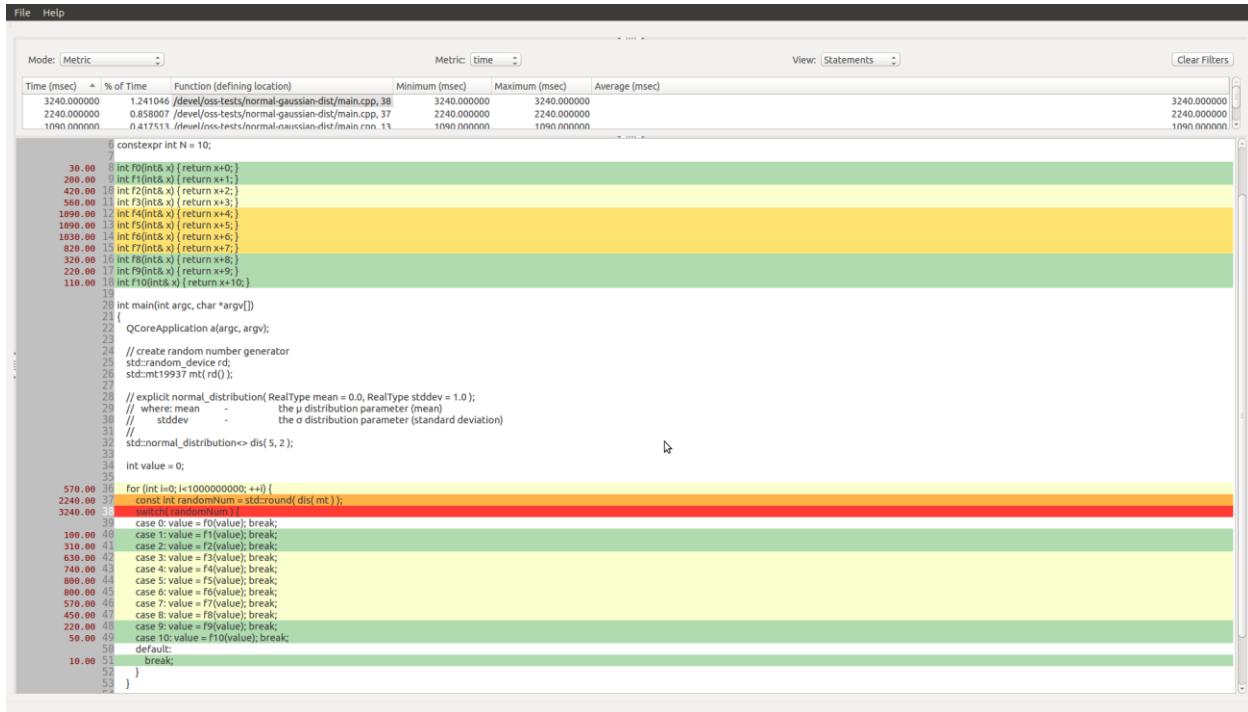


Figure 24 - Source-Code View with Metric Value Annotations

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). 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` templated 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 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 ▼
37 for (int i=0; i<10000000000; ++i) {
38 ▼
39     const int randomNum = dis(mt);
40     switch( randomNum ) {
41         case 0: value = f0(value); break;
42         case 1: value = f1(value); break;
43         case 2: value = f2(value); break;
44         case 3: value = f3(value); break;
45         case 4: value = f4(value); break;
46         case 5: value = f5(value); break;
47         case 6: value = f6(value); break;
48         case 7: value = f7(value); break;
49         case 8: value = f8(value); break;
50         case 9: value = f9(value); break;
51         case 10: value = f10(value); break;
52     default:
53         break;
54     }
55 }

```

Figure 25 - 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 26).

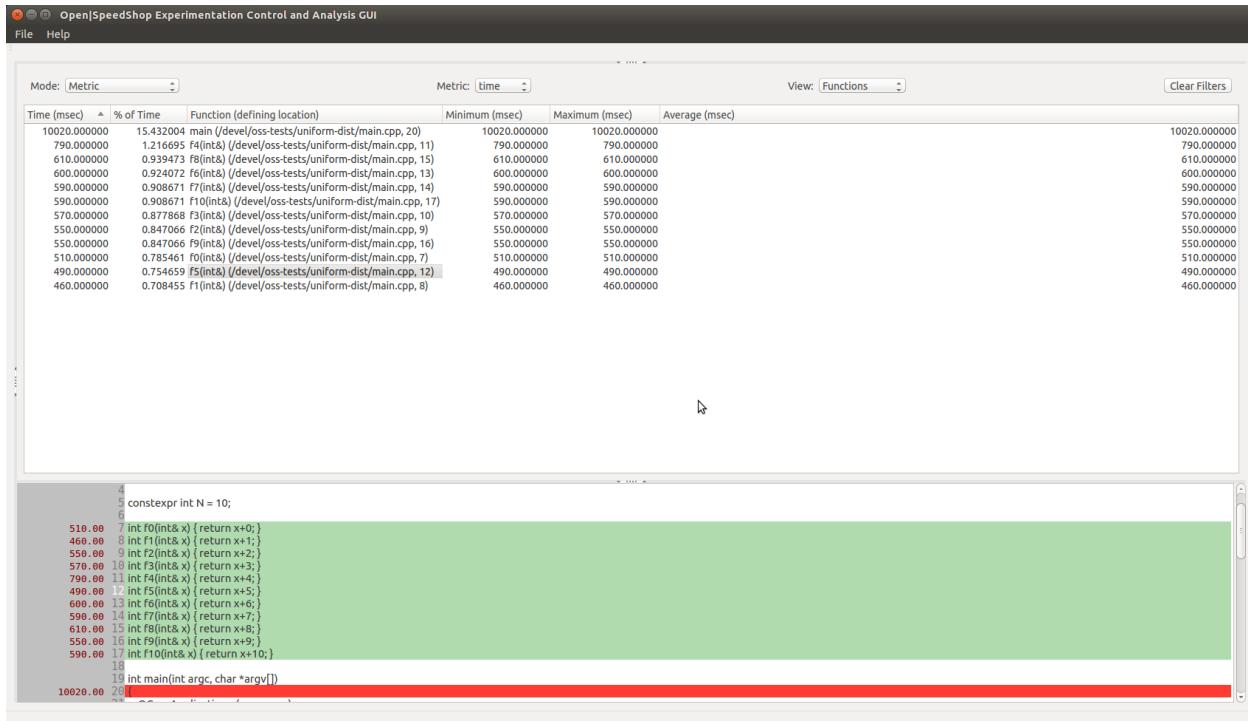


Figure 26 - Source-Code View with Metric Value Annotations – uniform distribution

### 9.1.4.2 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 27).

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 “pcsample” experiment and shown in Figures 22 and 23. 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 “pcsample” experiment (ref. Figure 28).

Unlike the “pcsample” 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. Figures 29 and 30).



Figure 27 - usertime experiment default view

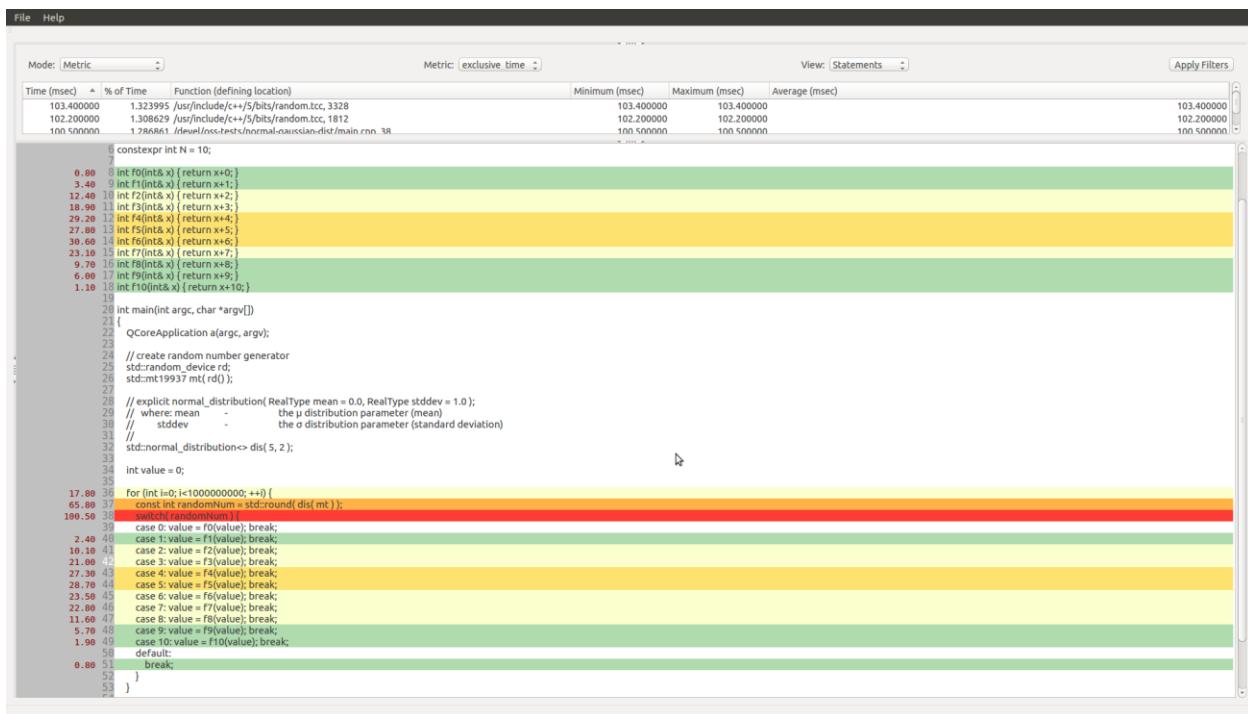


Figure 28 - Source-Code View with Metric Value Annotations (usertime)

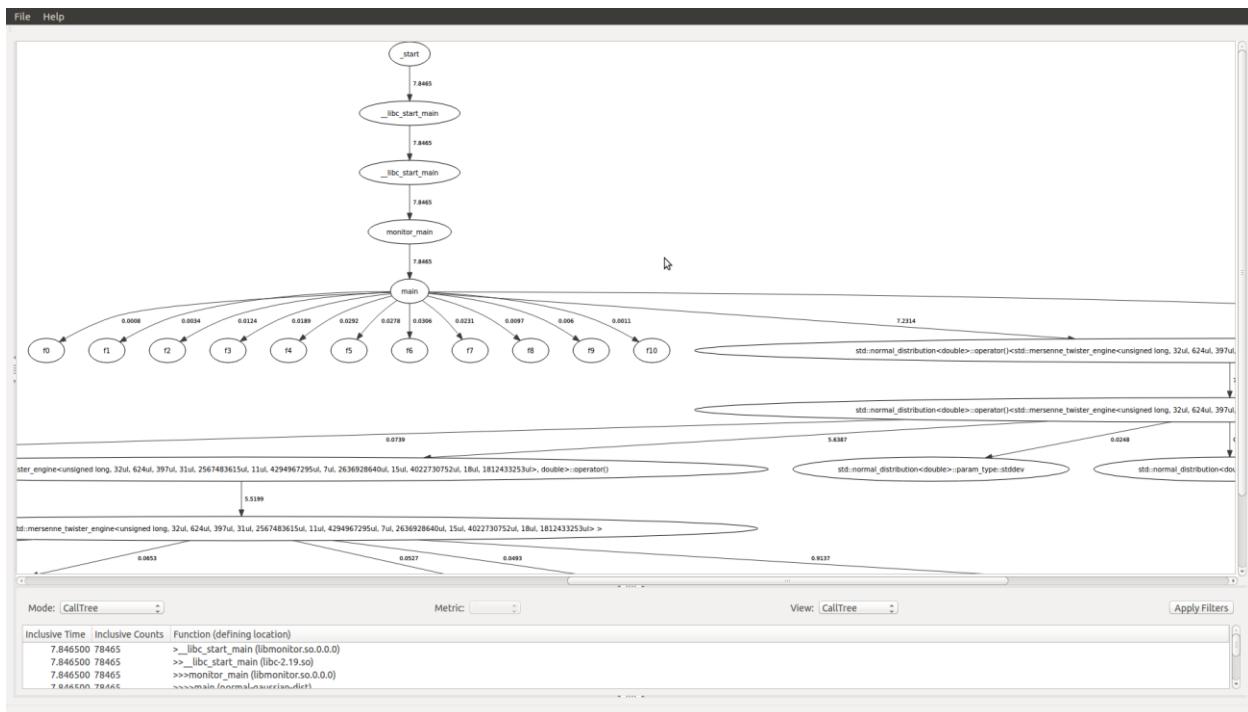


Figure 29 - Calltree Graph and Table Views

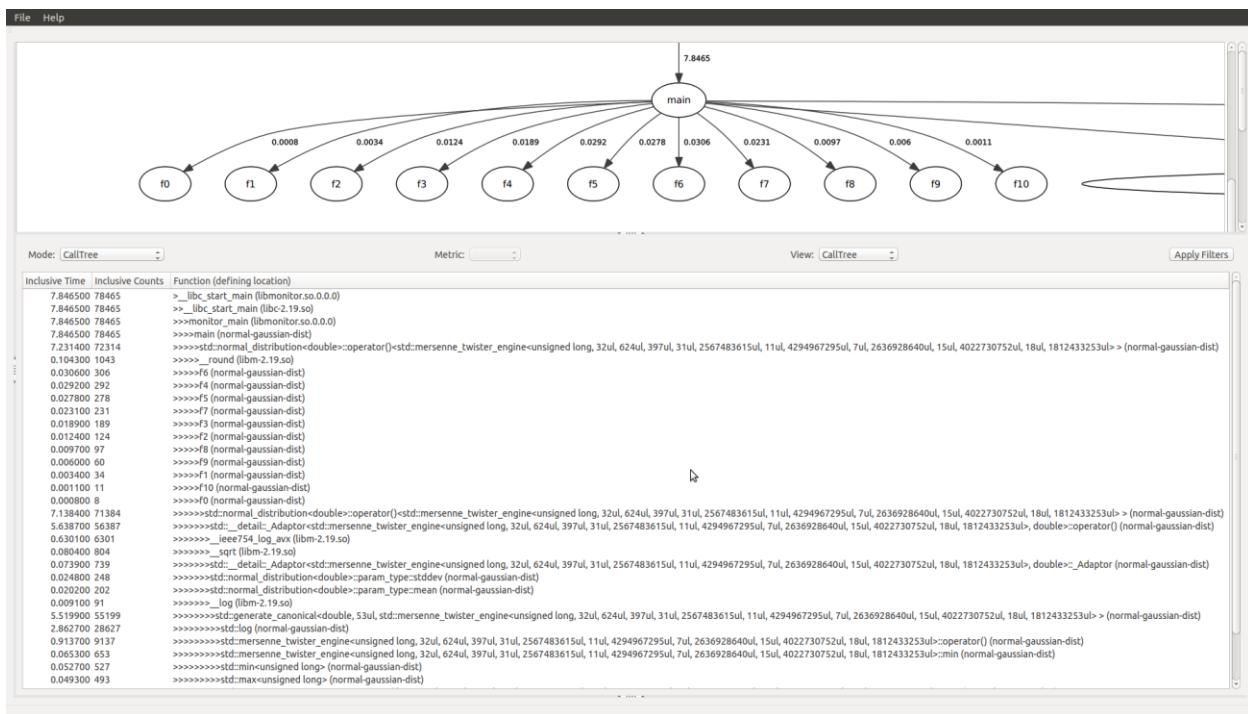


Figure 30 - Calltree Graph Zoomed into 'f1' to 'f10' Functions

Let's demonstrate a much more complex "usertime" experiment performed using the VASP application on an IBM Power 8 system (ref. Figure 31). Notice that since there are many functions profiled for the VASP application, the "Exclusive Time (Functions)" graph has compressed into a view without the labels for the individual function bars. However, let's examine the functions with the highest exclusive times by moving the cursor over the graph, hold the left-mouse button down and start dragging the mouse in a series of left-to-right movements until the graph expands to the point desired which shows the names of the functions with the highest exclusive times (ref. Figure 32).

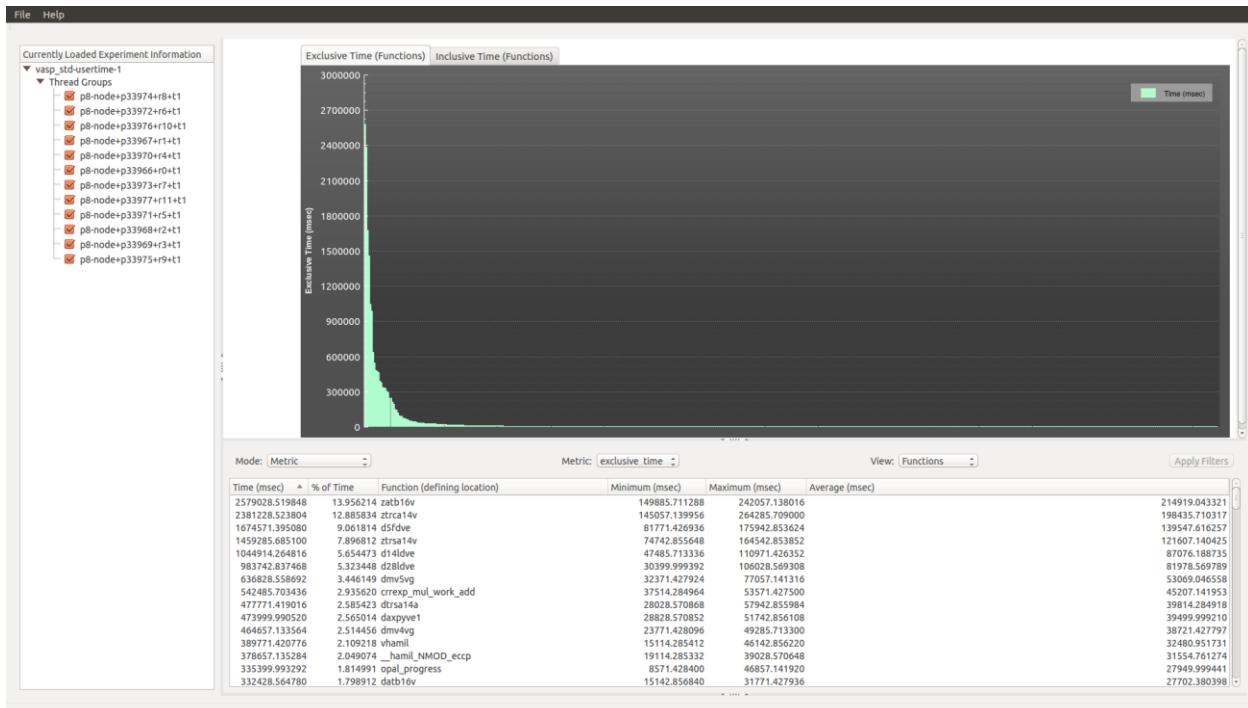


Figure 31 - usertime experiment default view (vasp)

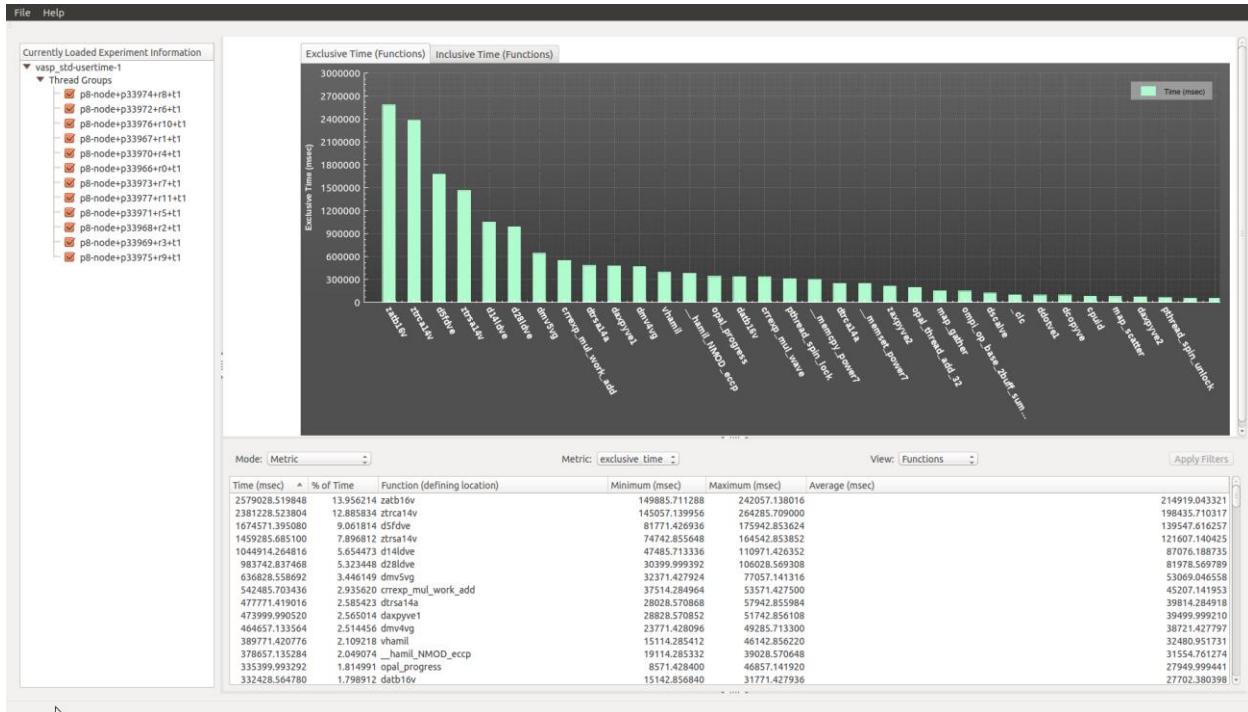


Figure 32 - usertime experiment default view (vasp functions with highest exclusive times)

The function-level view of the exclusive time metric values listed in Figure 32 are ordered by total accumulated exclusive times for all MPI ranks. The minimum, maximum and average function values for each function are also listed. A load balance view can be generated to see which MPI ranks are associated with the minimum, maximum and average function values for each function (ref. Figure 33). The load balance view is generated by selecting the “Load Balance” option in the “Mode” combo-box.



Figure 33 - usertime load balance view (vasp)

A comparison of the exclusive times can be generated by selecting the “Compare By Rank” option in the “Mode” combo-box (ref. Figure 34). This compare by rank view is sorting the functions by exclusive times for rank 0. Sorting by any of the other ranks can be achieved by clicking over the column header to cause the triangle icon to point upwards. The screenshots for the Compare By Rank views in which the table view is sorted by exclusive times for rank 1 (ref. Figure 35), rank 2 (ref. Figure 36) and rank 3 (ref. Figure 37). There is some slight change to the order of the functions as the Compare By Rank view is sorted by different ranks.

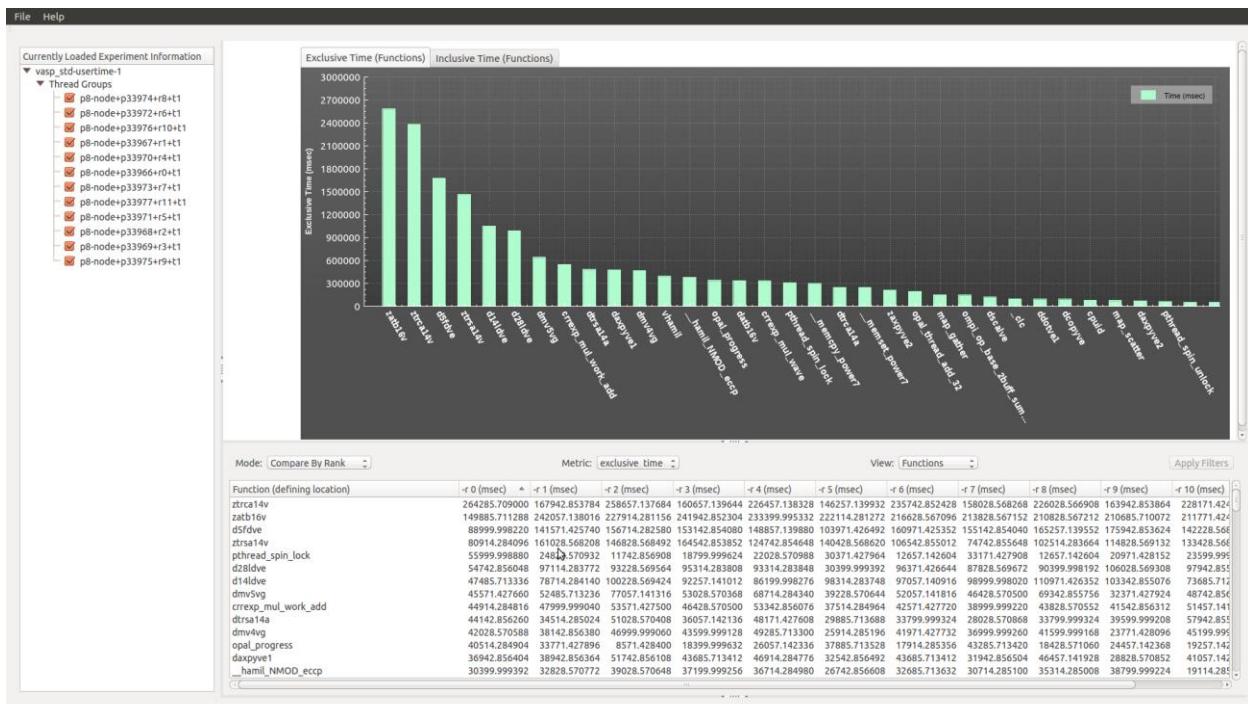


Figure 34 - usertime compare by rank view (vasp rank 0 sorting)

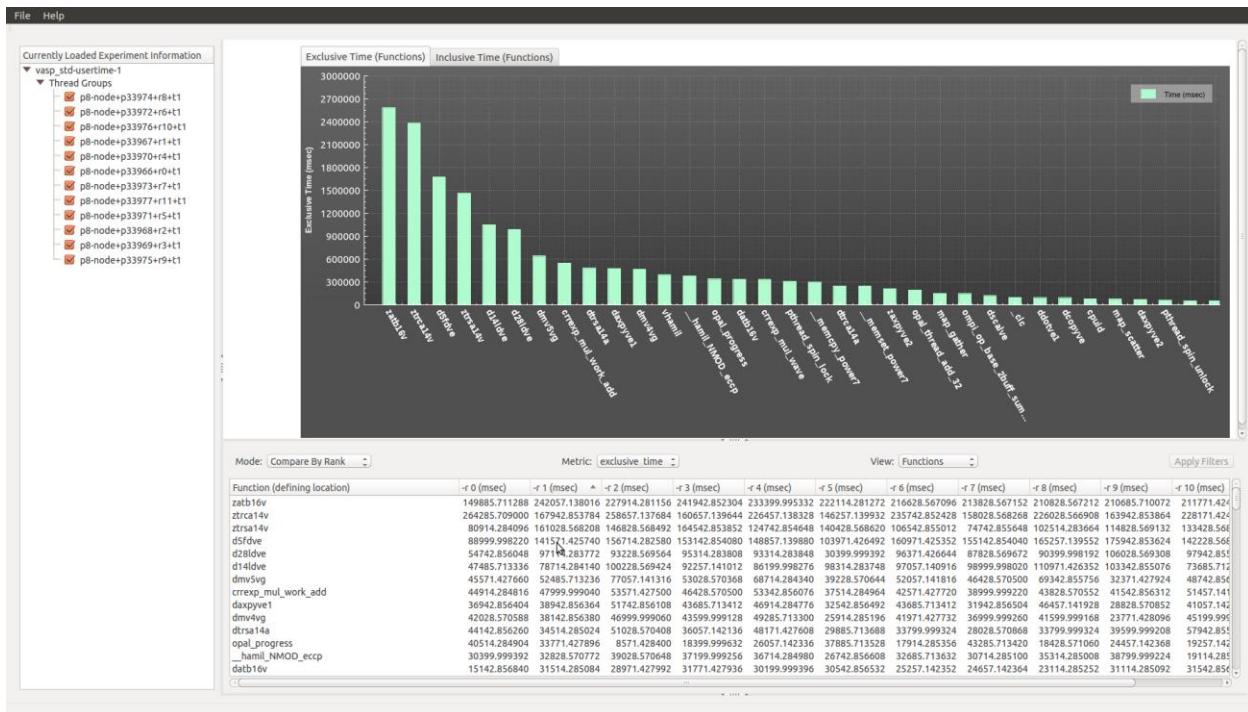


Figure 35 - usertime compare by rank view (vasp rank 1 sorting)



### 9.1.4.3 Using the OSS 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 38).

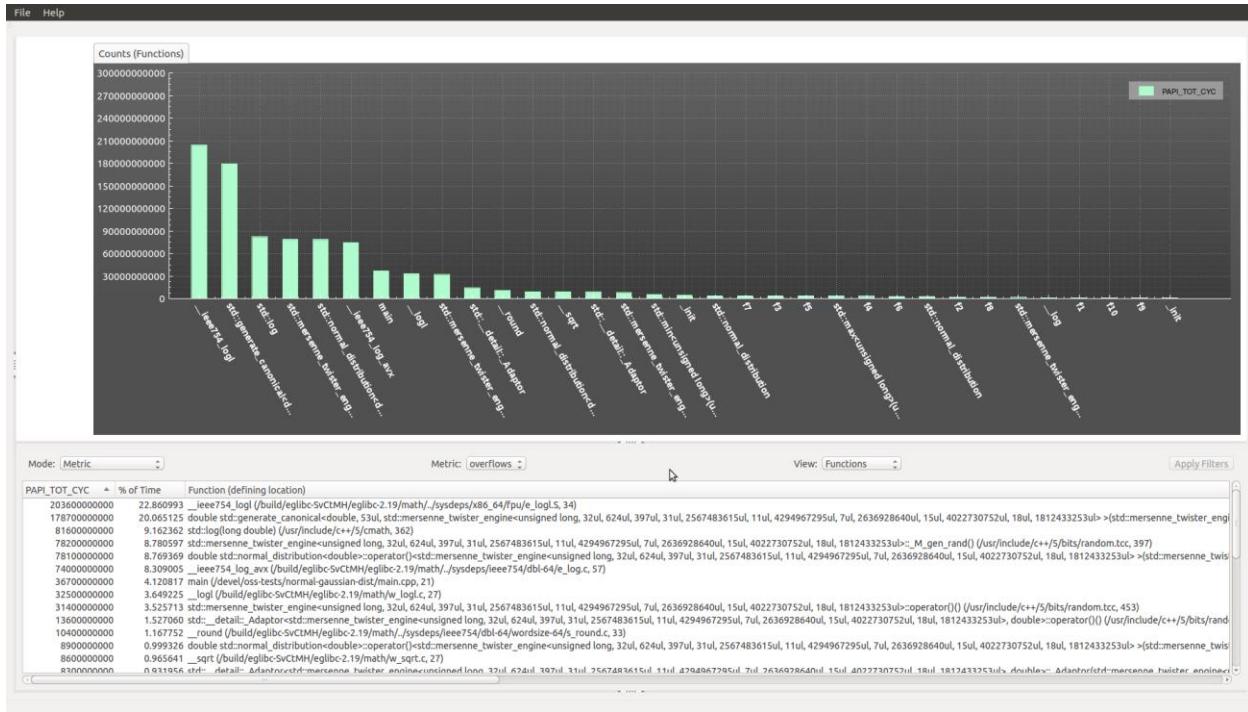


Figure 38 - hwc experiment default view

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 “pcsample” experiment and shown in Figures 22 and 23. 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 “pcsample” experiment (ref. Figure 39).

Observe metric annotations in the Source Code View in Figure 39. Since the “hwc” experiment only supports one PAPI event to be configured at a time, if the context-menu is activated in the Source Code View, it will list only the one PAPI event (see “PAPI\_TOT\_CYC” in Figure 39).

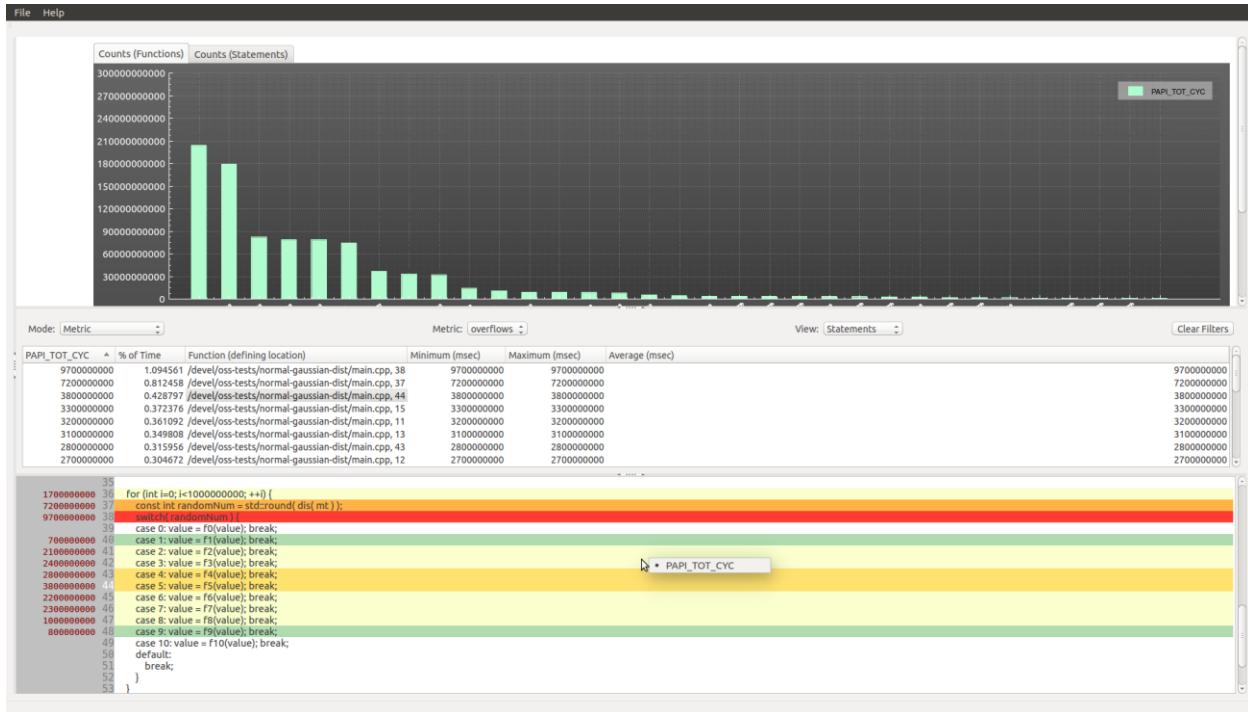


Figure 39 - hwc experiment detailed view

#### 9.1.4.4 Using the OSS 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 40).

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 41). 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 42). Figure 42 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 40 - hwctime experiment default view

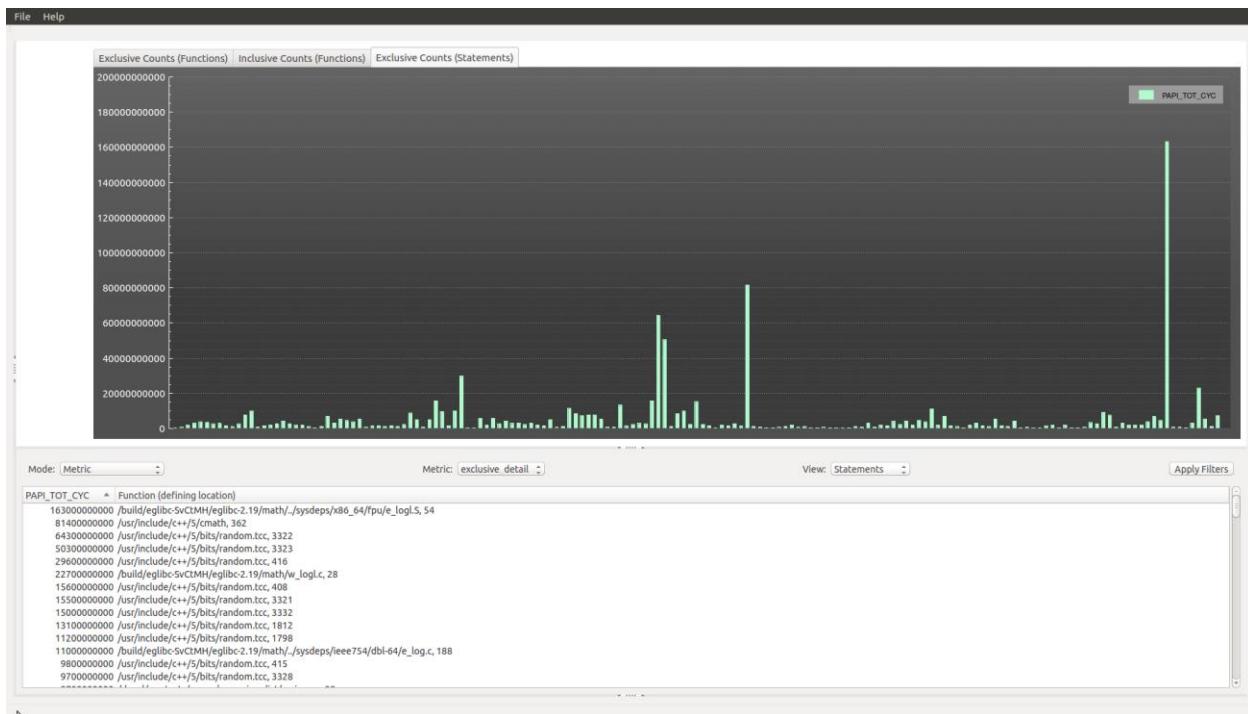


Figure 41 - hwctime exclusive-time metric statement-level view

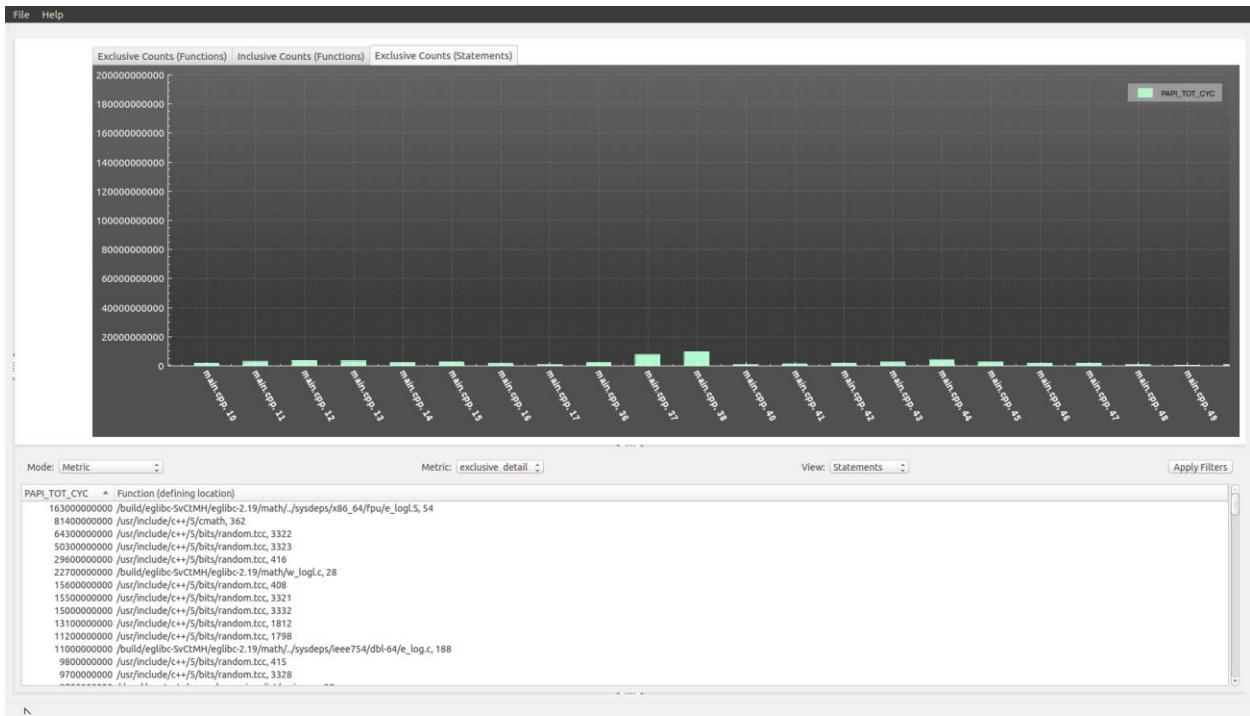


Figure 42 - zoomed graph view now showing x-axis labels

The screenshot in Figure 43 shows the metric annotations in the Source Code View. Like the “hwc” experiment, only one PAPI event can be configured for the experiment.

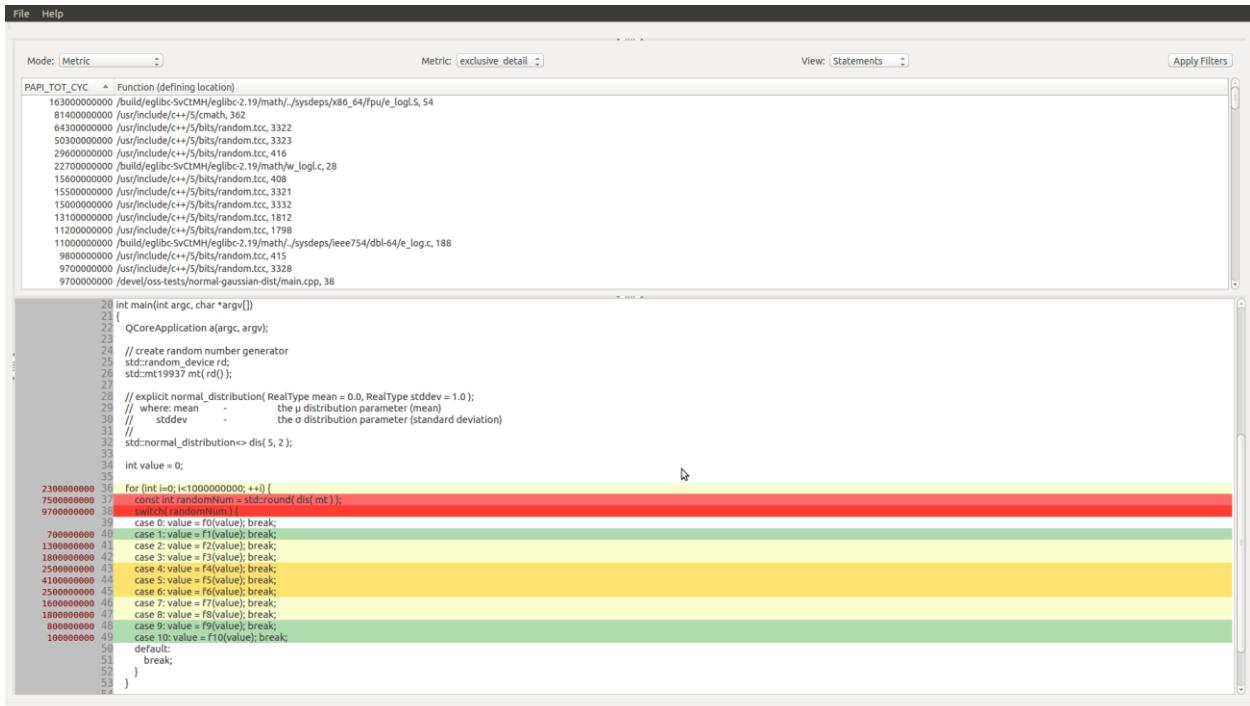


Figure 43 - HW counter values annotations for statement-view

#### 9.1.4.5 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 44).

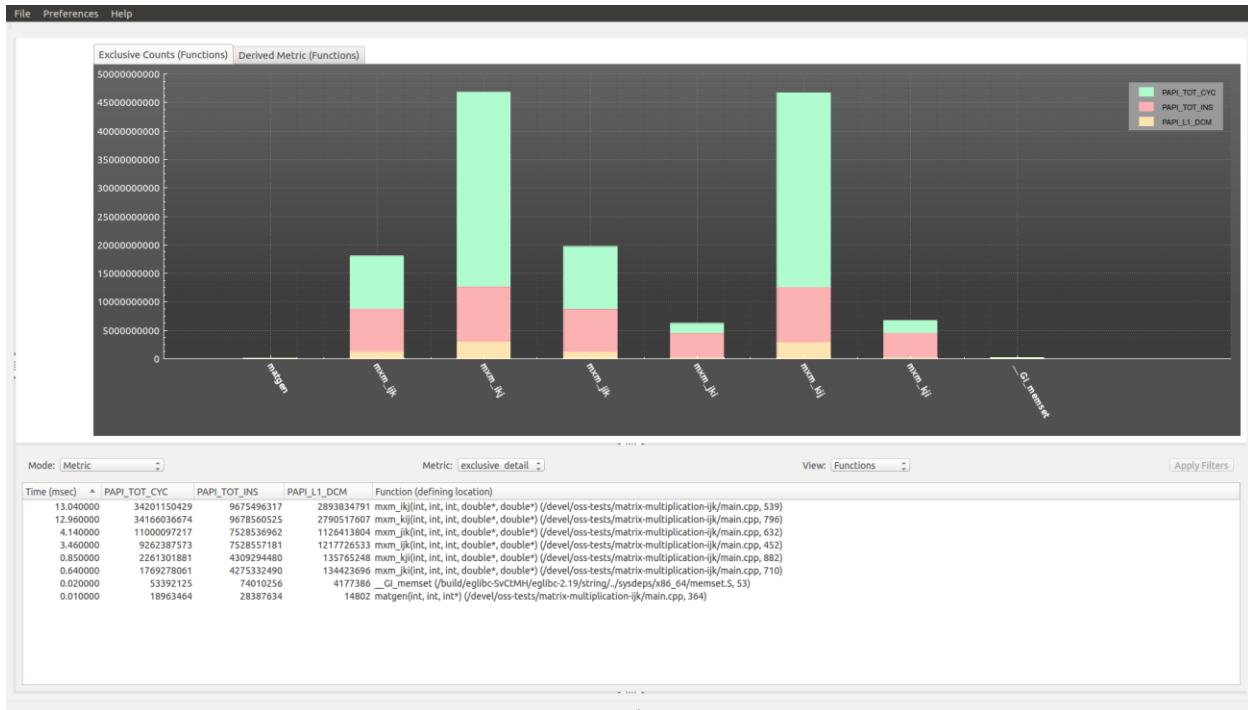


Figure 44 - hwcsamp experiment default view

For this case study 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](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 45a.

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: cbtfrun -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_ikj(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,882)
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 45a - 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 FOR loop variants that show the lowest CPU cycles, instructions issued and Level 1 data cache misses (ref. Figure 44). Figure 44 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 47).

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 44 or in Figure 45a, 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. The IPC values shown in Table 1 can be matched to the “Instructions Per Cycle” column in the Derived Metric mode view (ref. Figure 46).

Metric	IJK	IKJ	JIK	JKI	KIJ	KJI
PAPI_TOT_INS	7528557181	9675496317	7528536962	4275332490	9678560525	4309294480
PAPI_TOT_CYC	9262387573	34201150429	11000097217	1769278061	34166036674	2261301881
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 45b. 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().

```

double mmx_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 ( int j = 0; j < n3; j++ )
    {
        for ( int k = 0; k < n2; k++ )
        {
            for ( int 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;
}

double mmx_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 ( int k = 0; k < n2; k++ )
    {
        for ( int j = 0; j < n3; j++ )
        {
            for ( int 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;
}

```

Figure 45b - JKI / KJI variant source-code

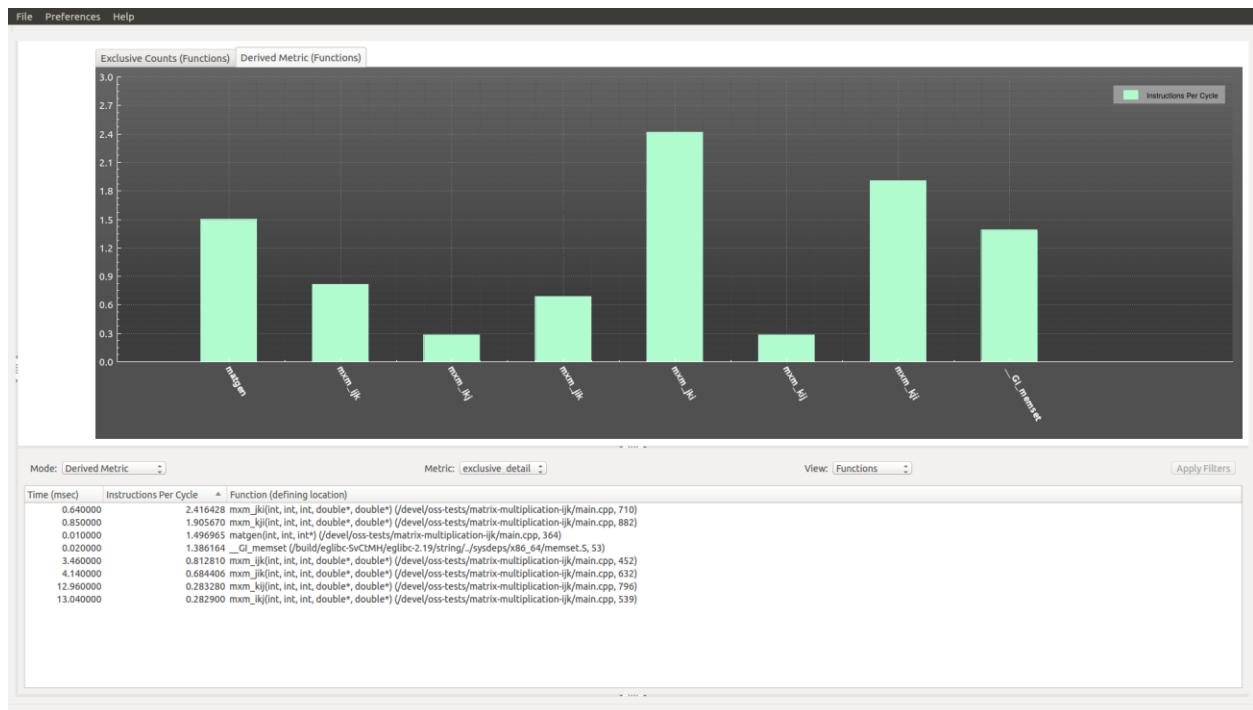


Figure 46 - Derived Metric Mode View

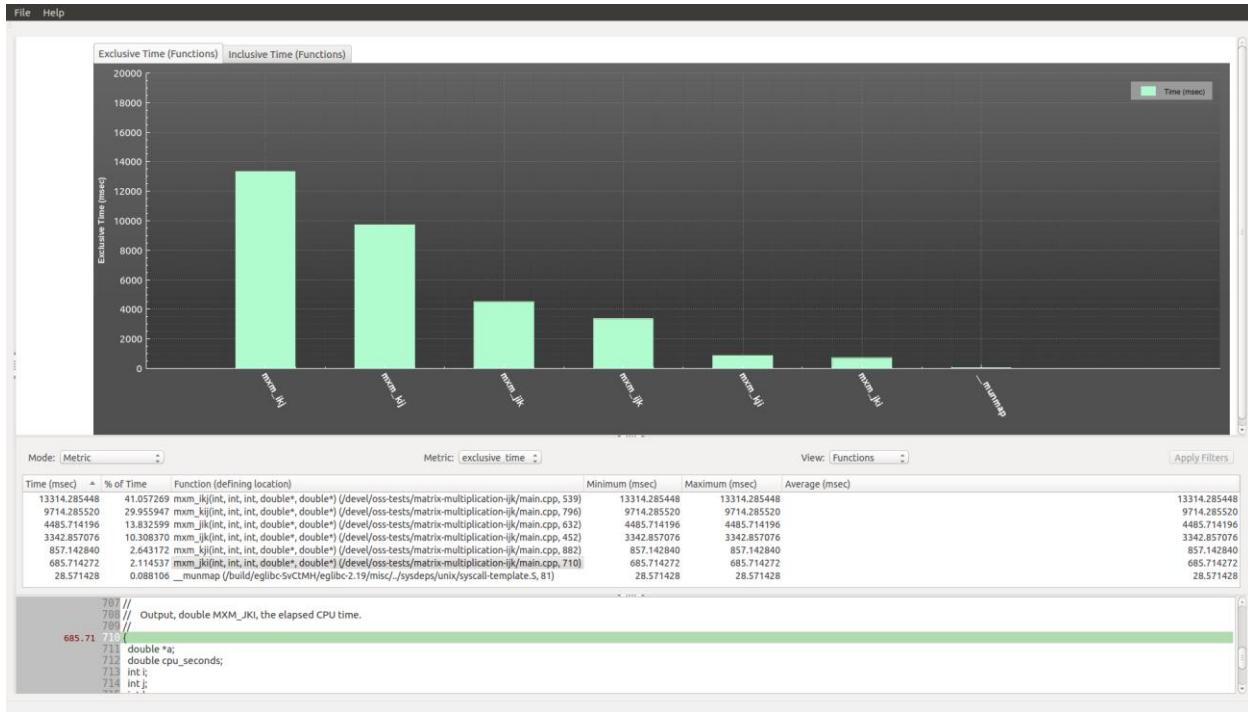


Figure 47 - exclusive times for nested FOR loop variants using the usertime experiment

#### 9.1.4.6 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 48). Currently there is no graph generated in the Metric Plot View.

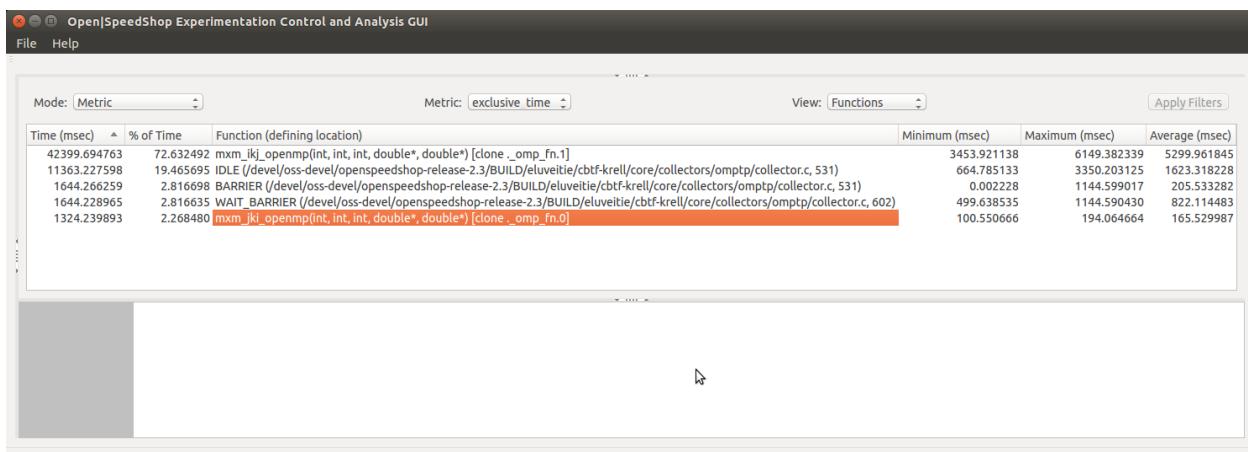


Figure 48 - omptp experiment default view

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 49).

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 50).

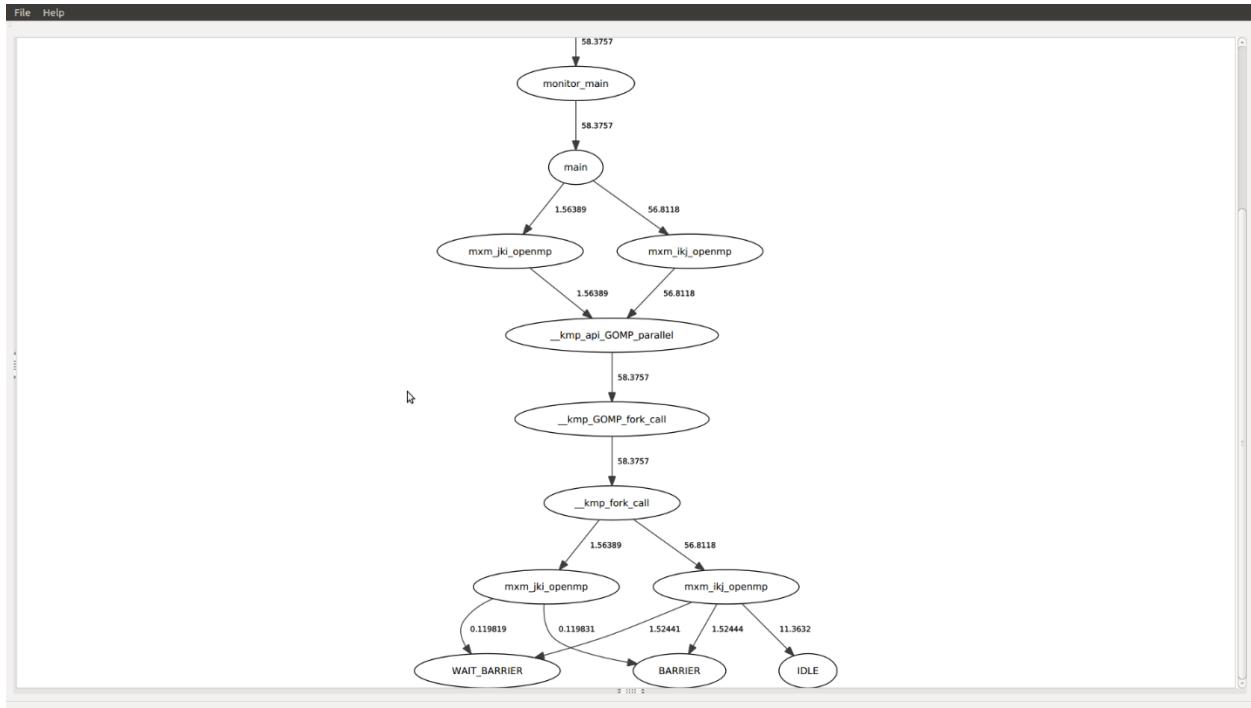


Figure 49 - calltree graph showing caller-callee relationships

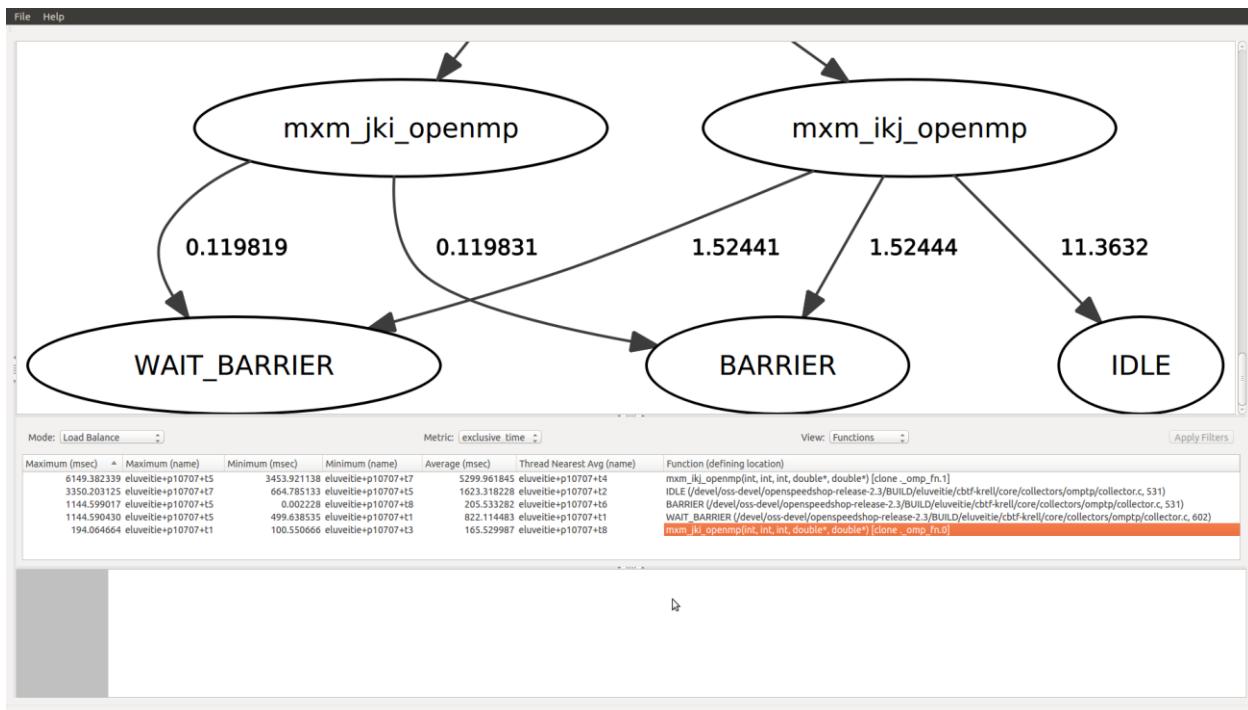


Figure 50 - omptp load balance view

#### 9.1.4.7 Using the OSS 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 51) showing detailed memory event information. Figure 52 shows the source-code in which all memory allocations are leaked (10 total) and Figure 53 shows the source-code in which half the memory allocations are leaked (5 total).

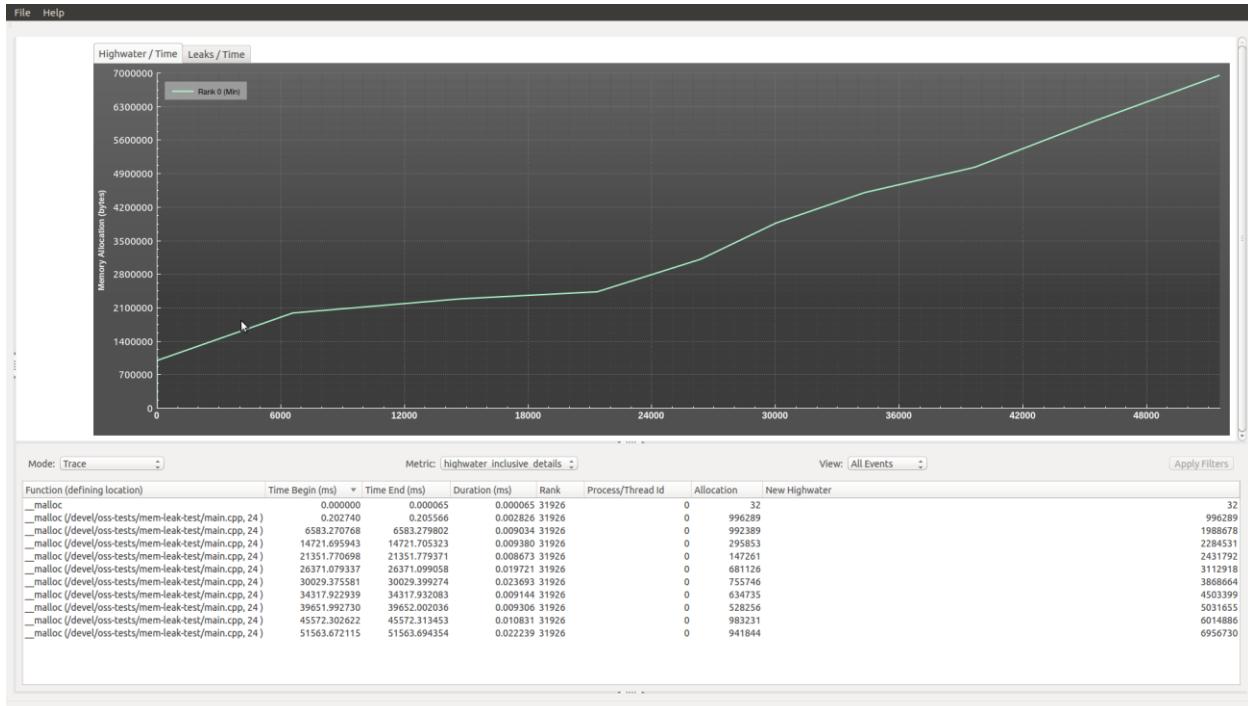


Figure 51 - mem experiment default view

```

for (int i=0; i<10; ++i) {
    char* buffer = (char *) malloc( dis(mt) );

    usleep( sleepdis(mt)*1000*1000 );
}

```

Figure 52 – 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 53 – source-code with 5 memory leaks

Observe the Metric Table View in Figure 54 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 54 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. Line 24 contains a call to the “malloc” memory allocation function.

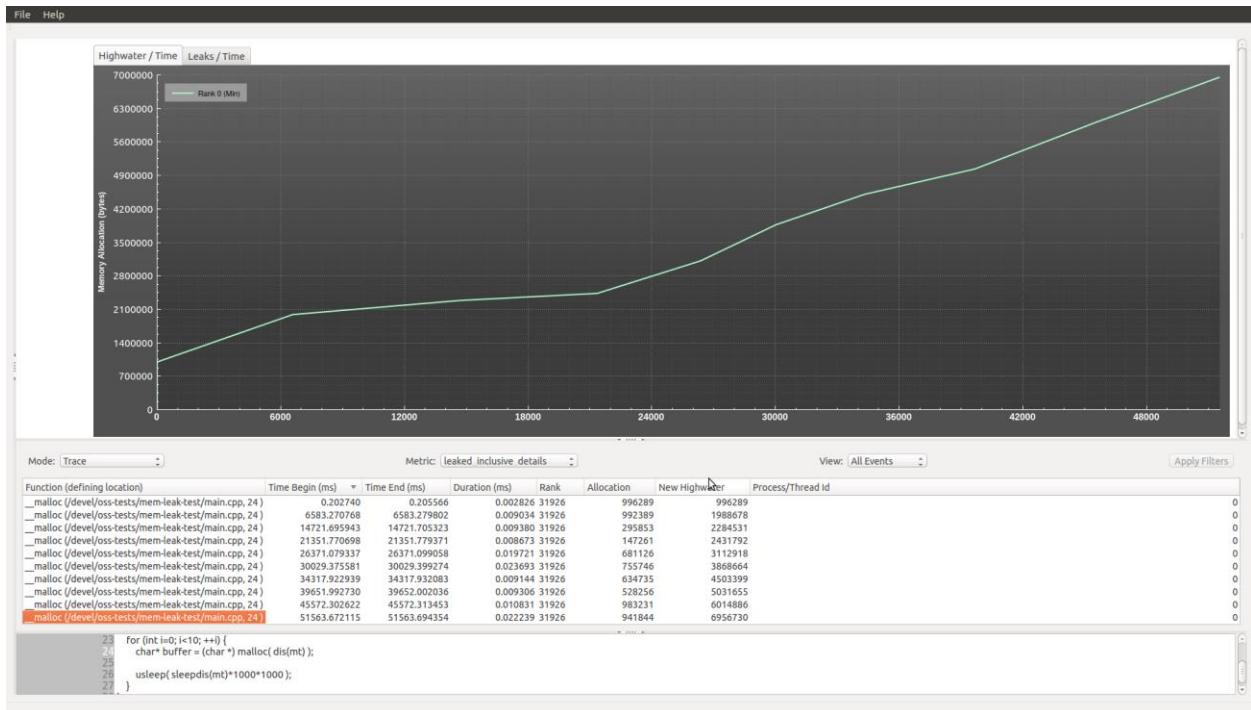


Figure 54 – ten memory leaks associated with Figure 52 source-code

Observe the Metric Table View in Figure 55 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 55 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. Line 27 also contains a call to the “malloc” memory allocation function.



Figure 55 – five memory leaks associated with Figure 53 source-code

#### 9.1.4.8 Using the OSS 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 56).

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 57) and Compare By Rank (ref. Figure 58) views. These are generated by selecting the “Compare By Process” or “Compare By Rank” option respectively 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 59).

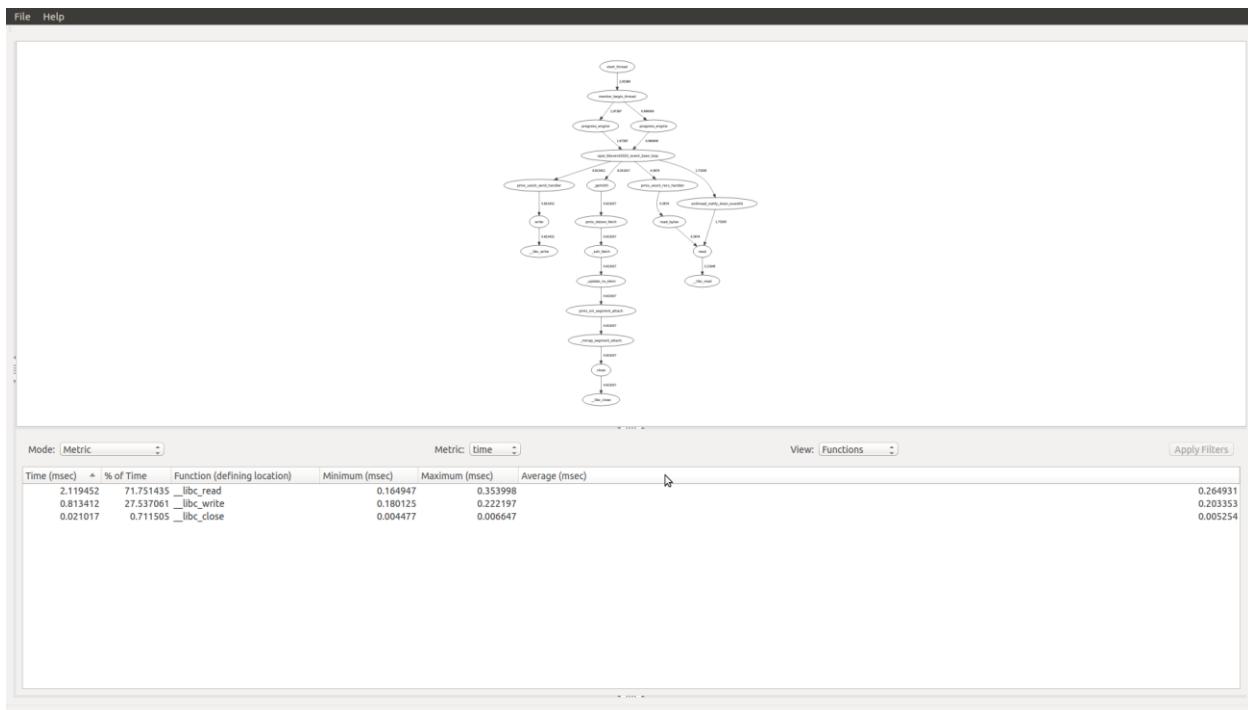


Figure 56 - time metric view (with calltree graph)

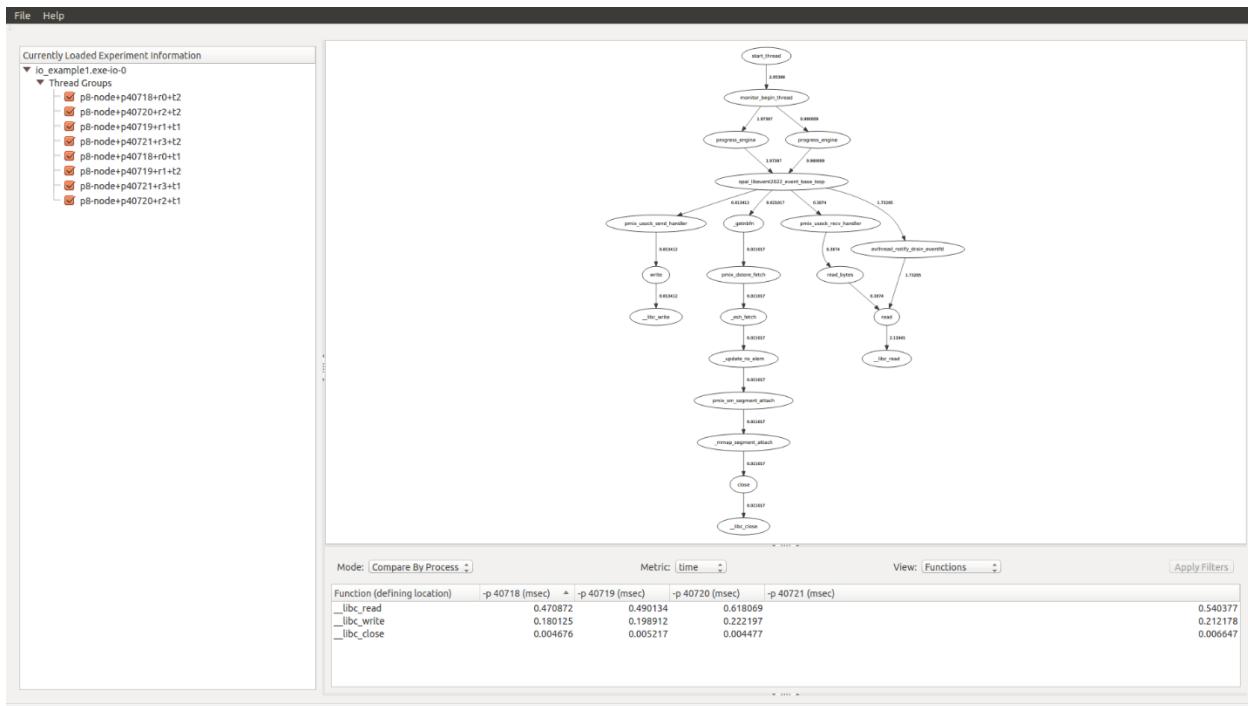


Figure 57 - compare by process view (with calltree graph)

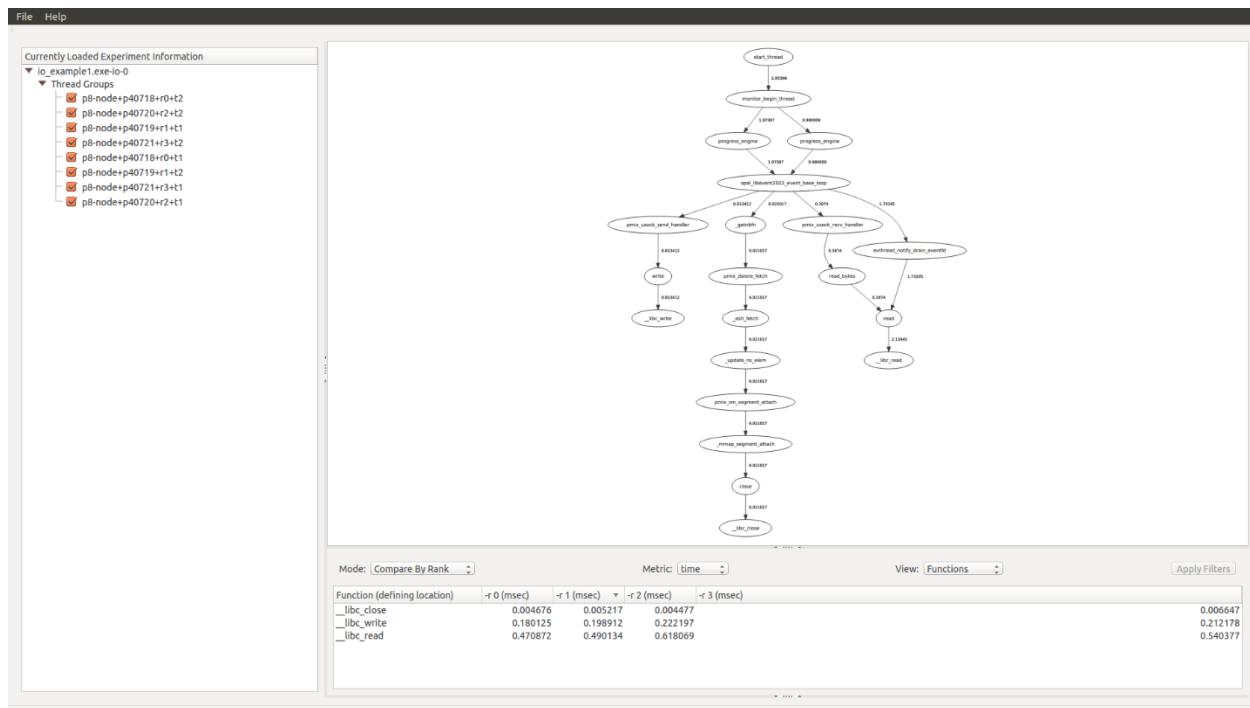


Figure 58 - compare by rank view (with calltree graph)

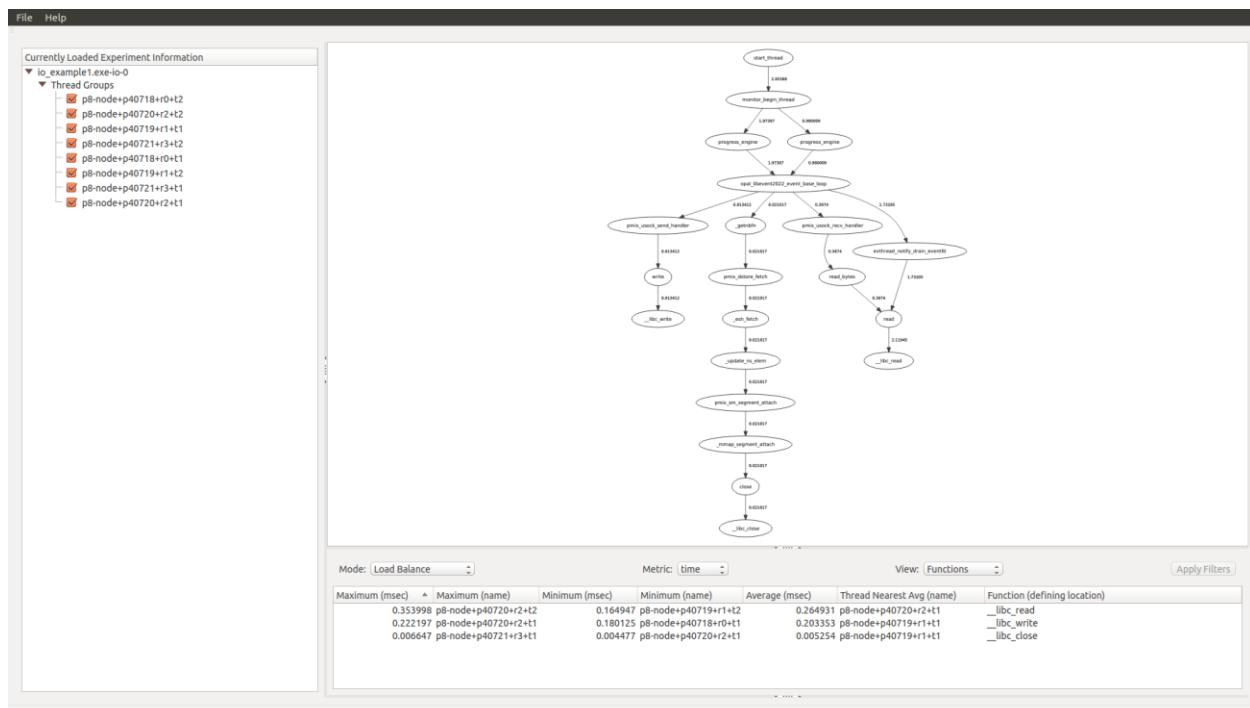


Figure 59 - load balance view (with calltree graph)

### 9.1.4.9 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 60).

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 “ossioip” convenience script is executed.

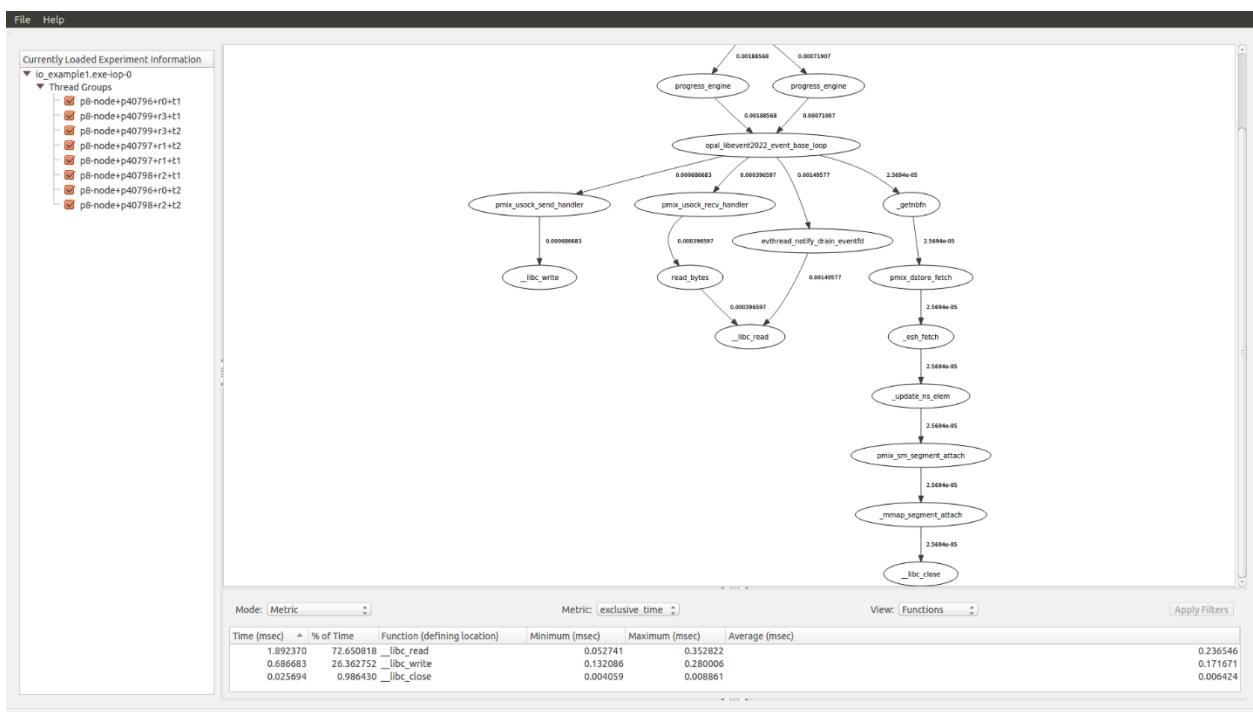


Figure 60 - iop experiment time metric view (with calltree graph)

Additional views of possible interest to help point out imbalance of processing between processes or ranks are the Compare By Process (ref. Figure 61) and Compare By Rank (ref. Figure 62) 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 63).

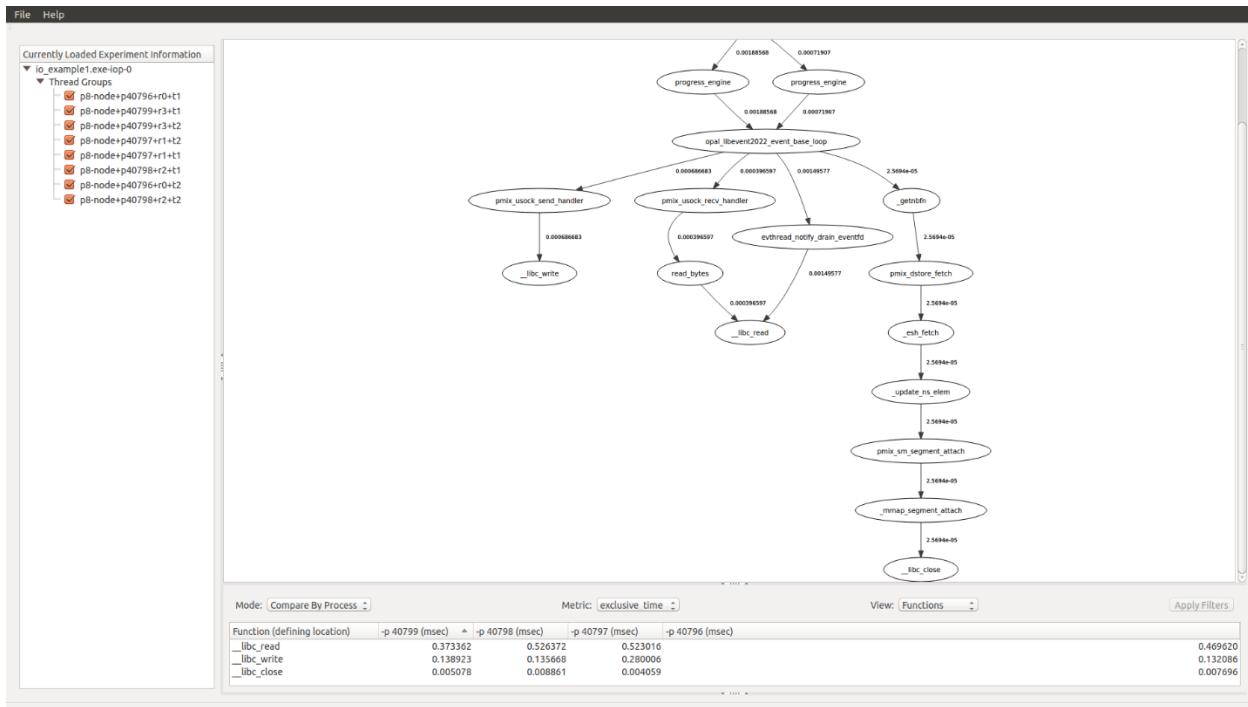


Figure 61 - iop experiment compare by process view (with calltree graph)

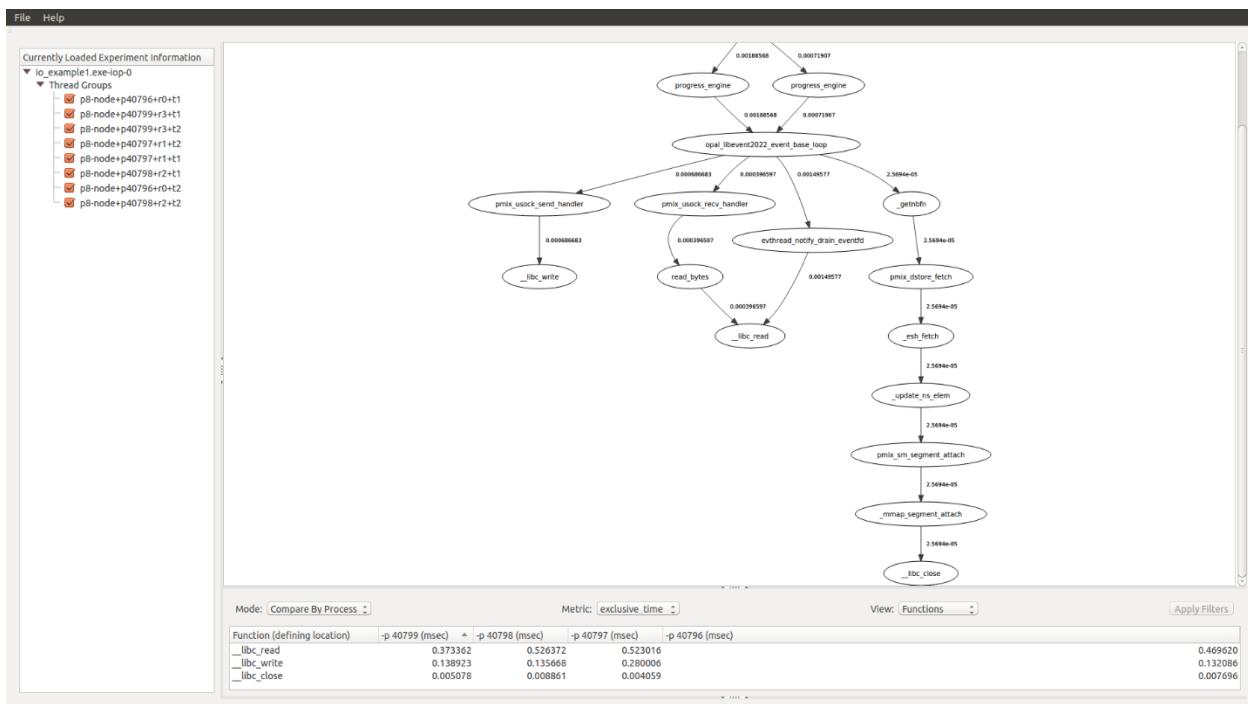


Figure 62 – iop experiment compare by rank view (with calltree graph)

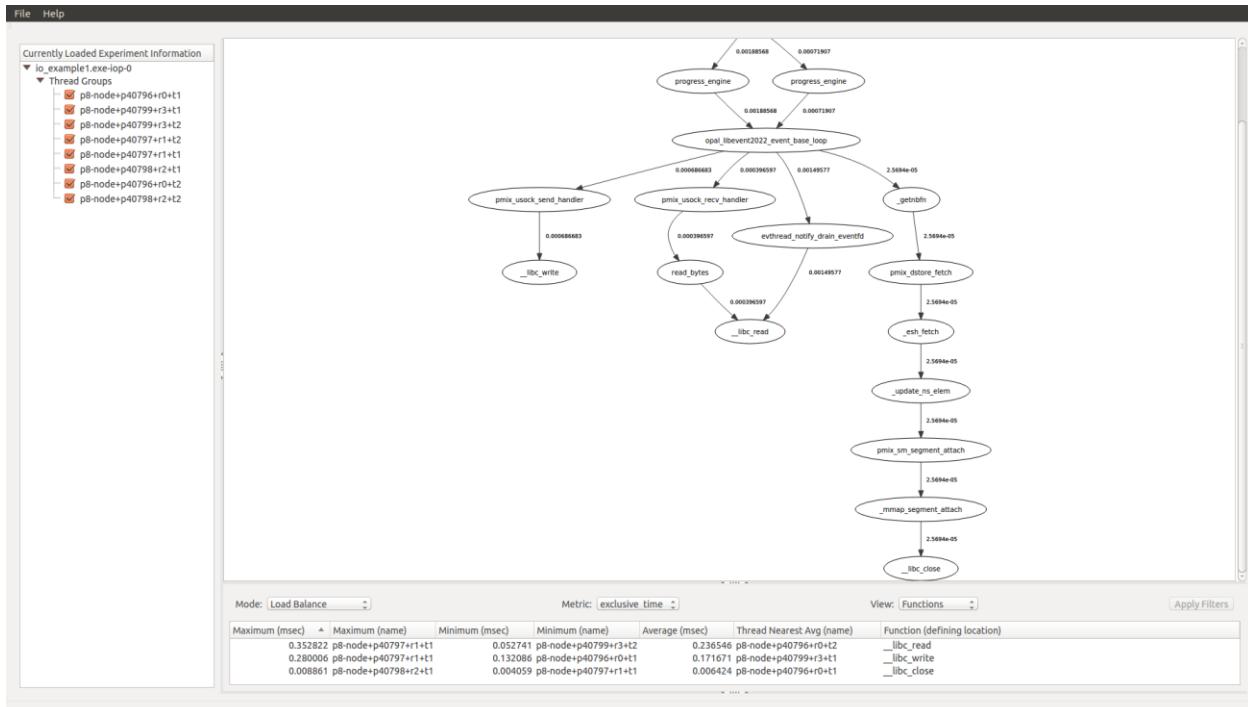


Figure 63 – iop experiment load balance view (with calltree graph)

#### 9.1.4.10 Using the OSS 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 determined.

Upon loading the “iot” experiment, the default view appears showing the I/O event timeline and exclusive time metric values for the functions view (ref. Figure 64). 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 65) and Compare By Rank (ref. Figure 66) 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 67).

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 68).

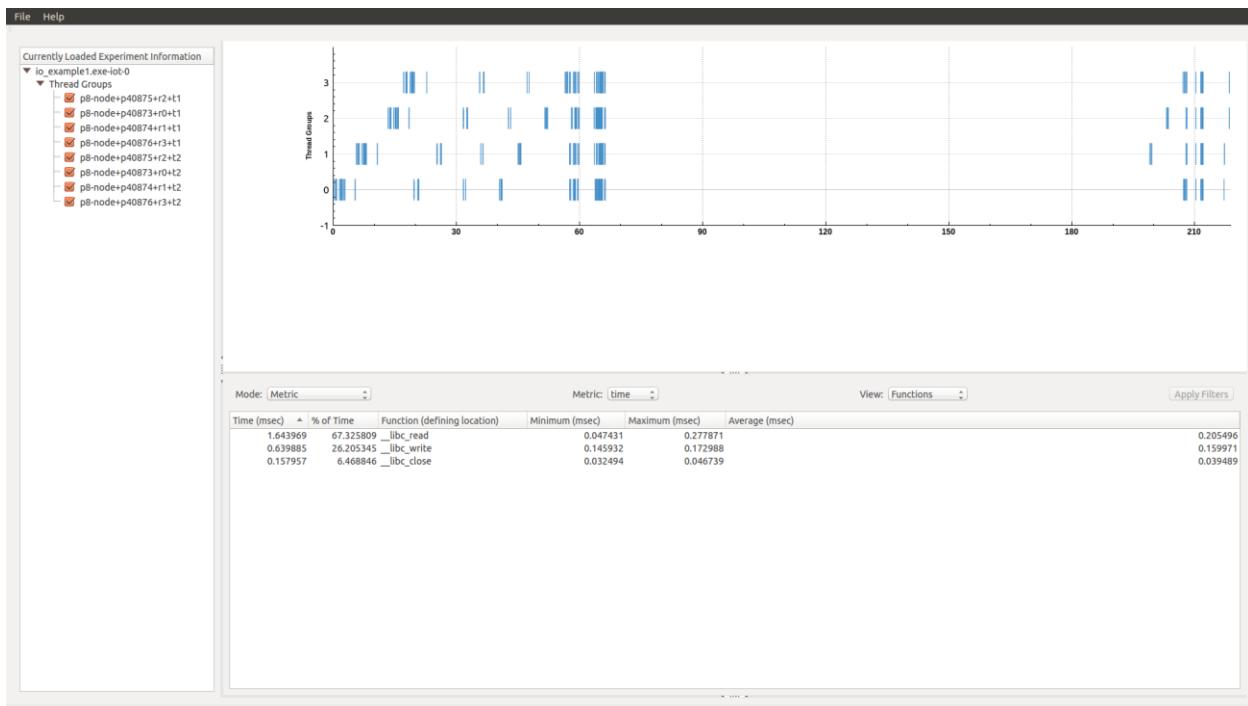


Figure 64 - iot experiment default view

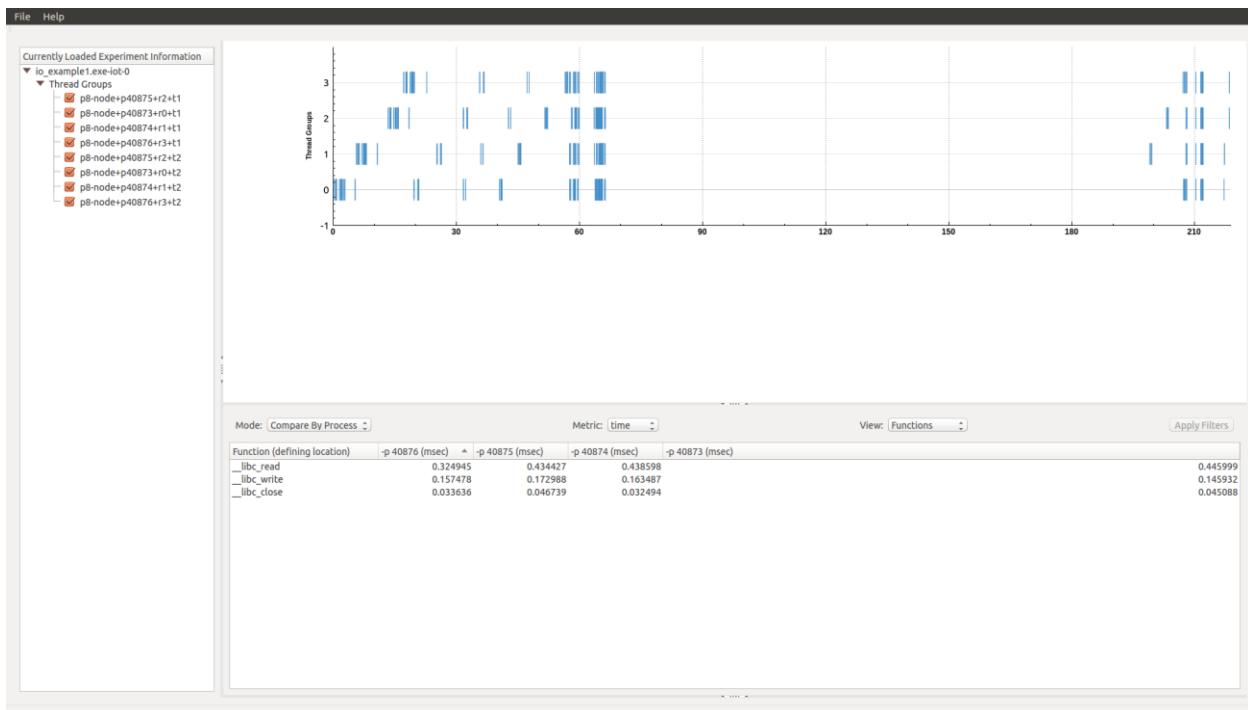


Figure 65 - iot experiment compare by process view

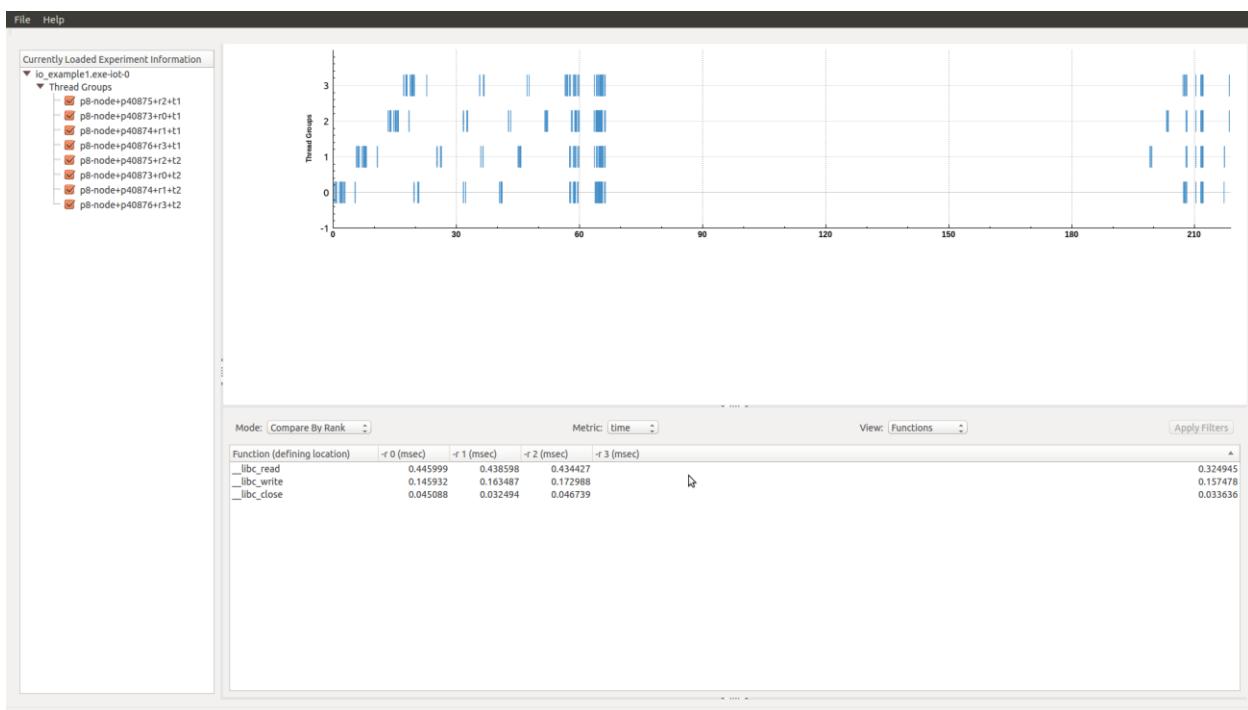


Figure 66 - iot experiment compare by rank view

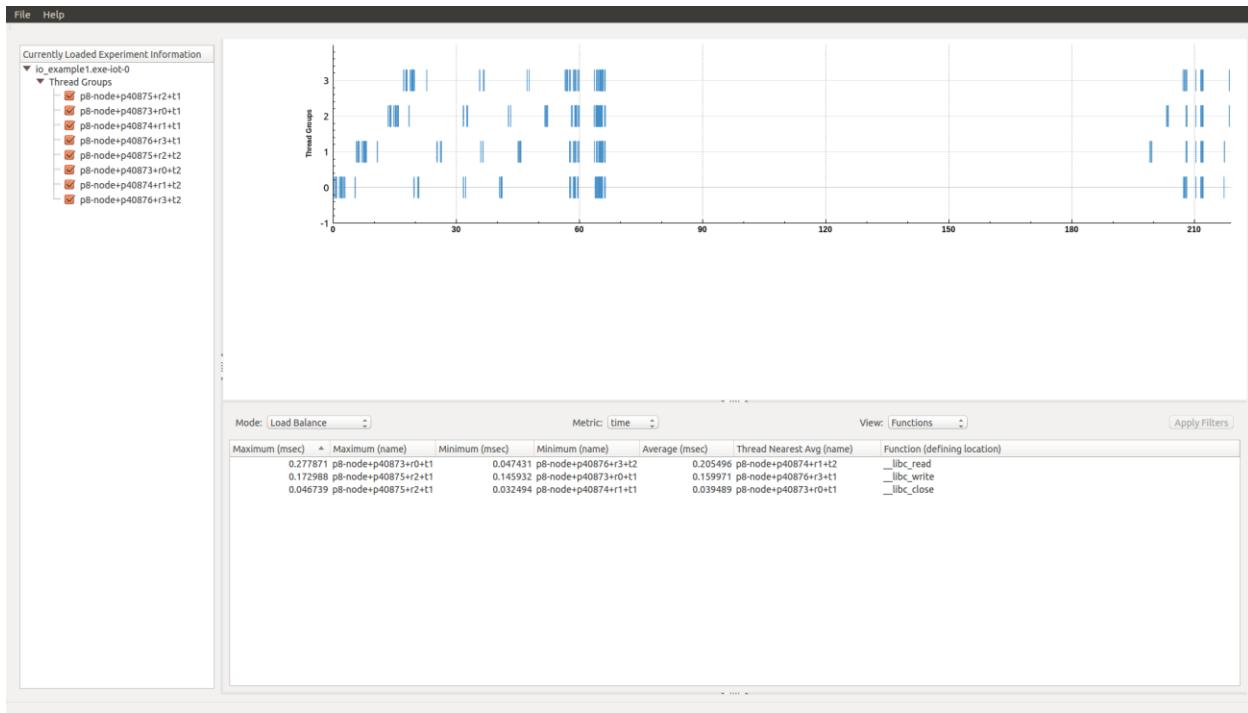


Figure 67 - iot experiment load balance view

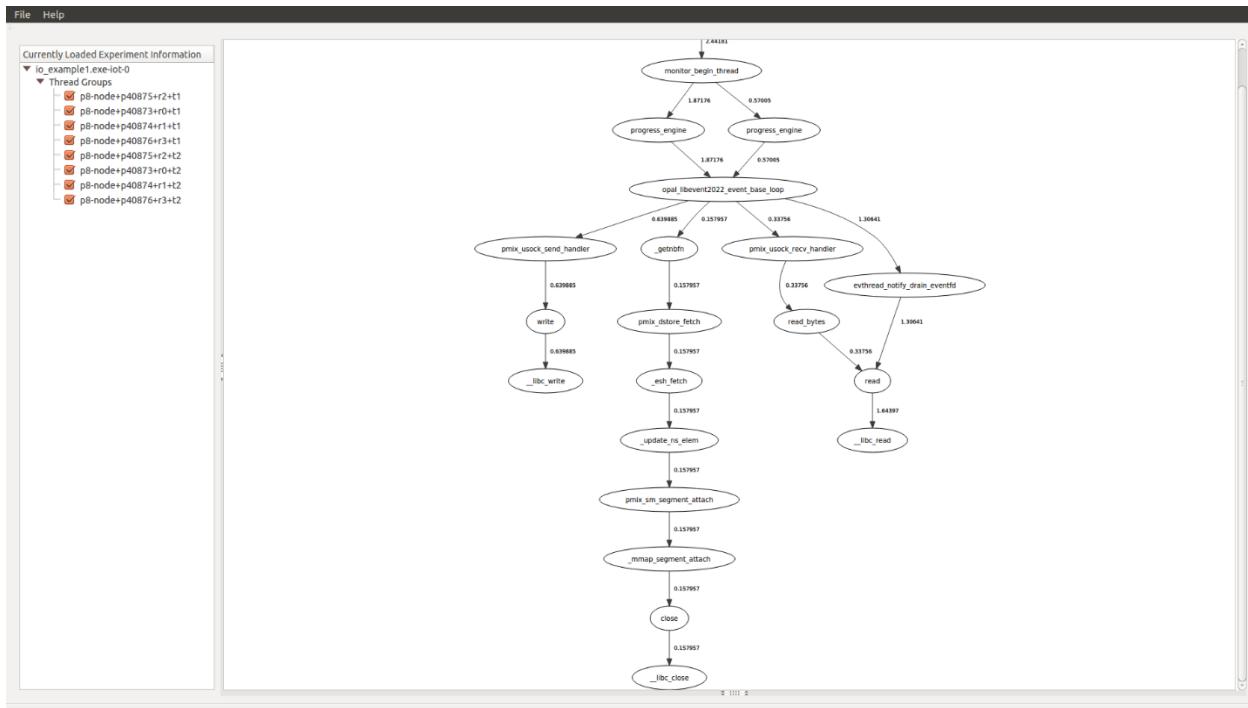


Figure 68 - iot experiment calltree graph

### 9.1.4.11 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 69).

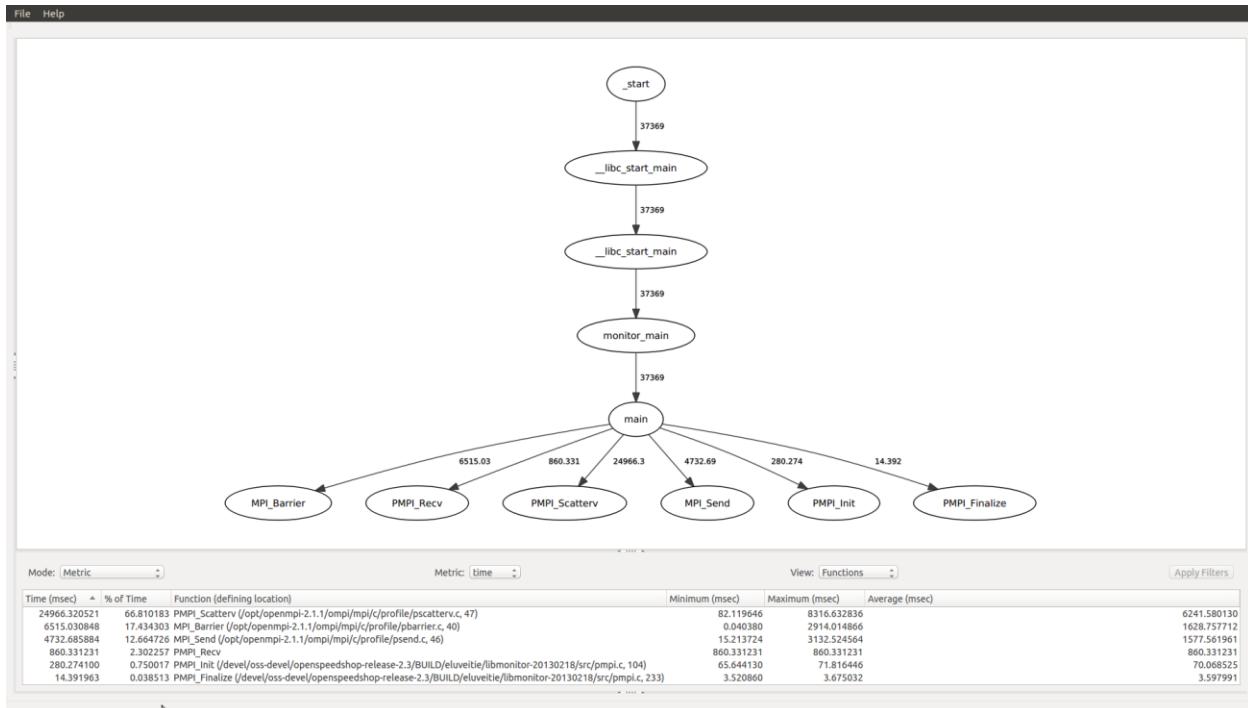


Figure 69 - mpi experiment default view (with calltree graph)

Additional views of possible interest to help point out imbalance of processing between processes or ranks are the Compare By Process (ref. Figure 70); Compare By Rank (ref. Figure 71) views; and Compare (ref. Figure 72) views. These are generated by selecting the “Compare By Process”, “Compare By Rank” or “Compare” option respectively 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 73).

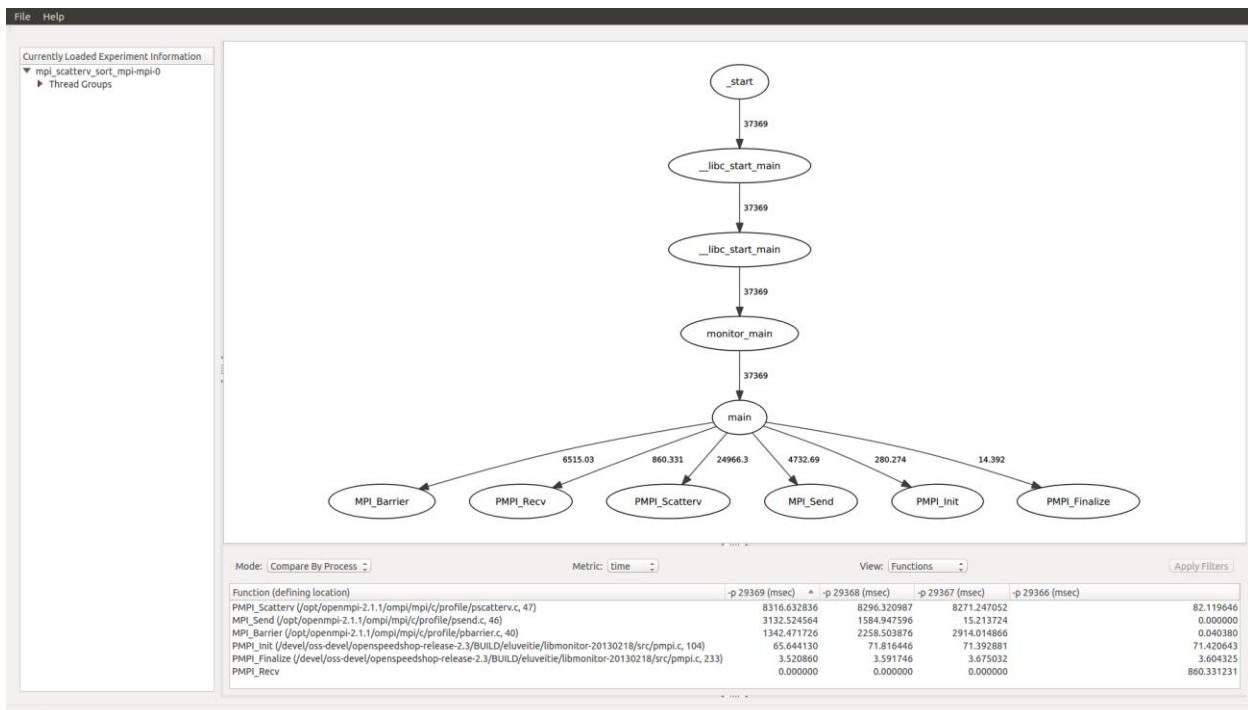


Figure 70 - compare by process view (with calltree graph)

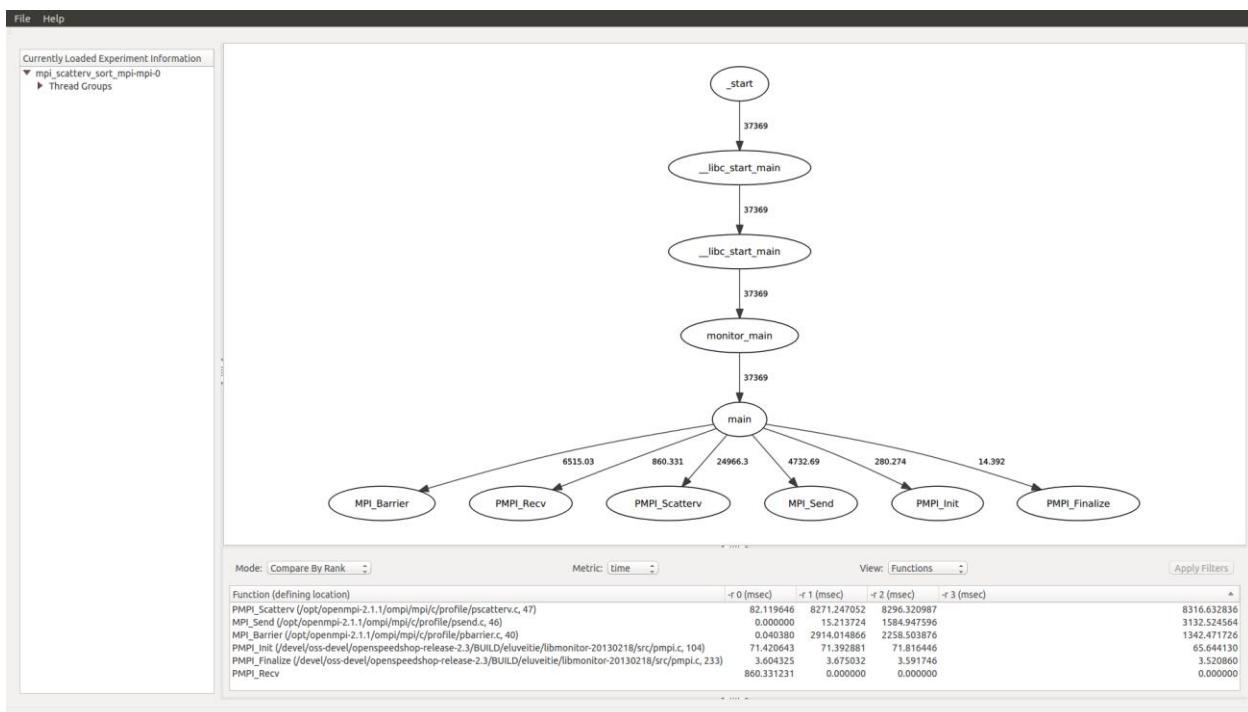


Figure 71 - compare by rank view (with calltree graph)

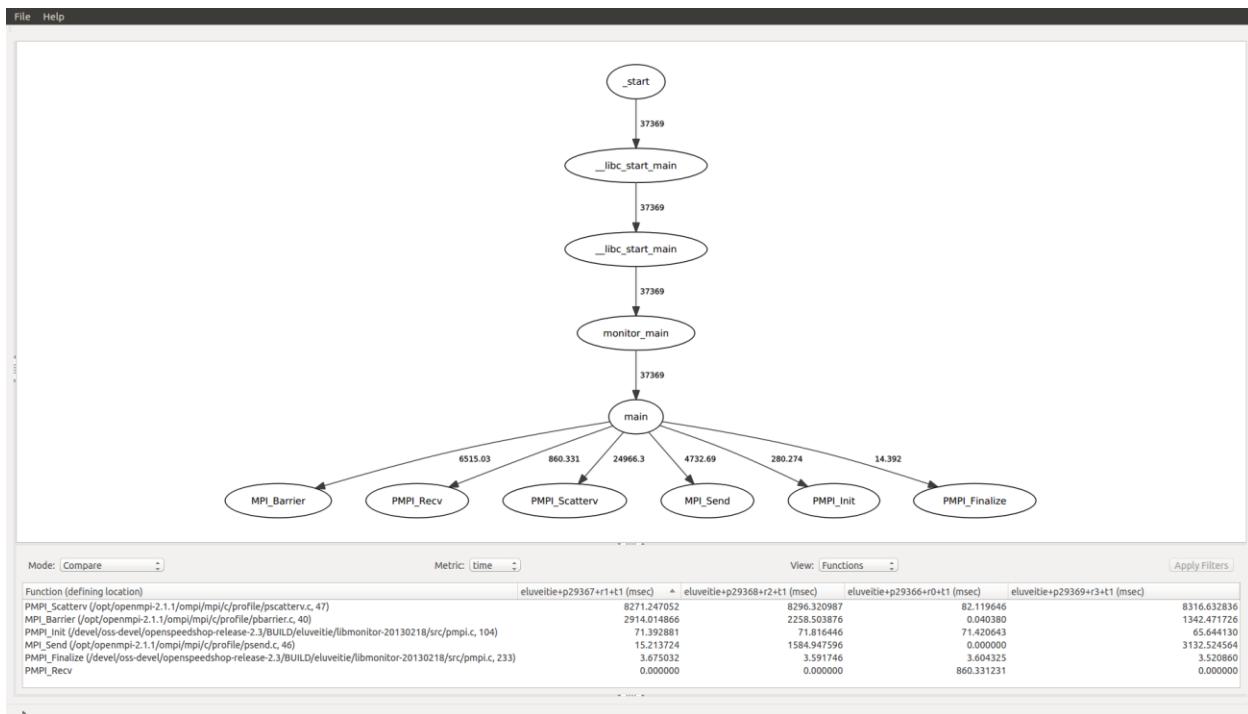


Figure 72 - compare view (with calltree graph)

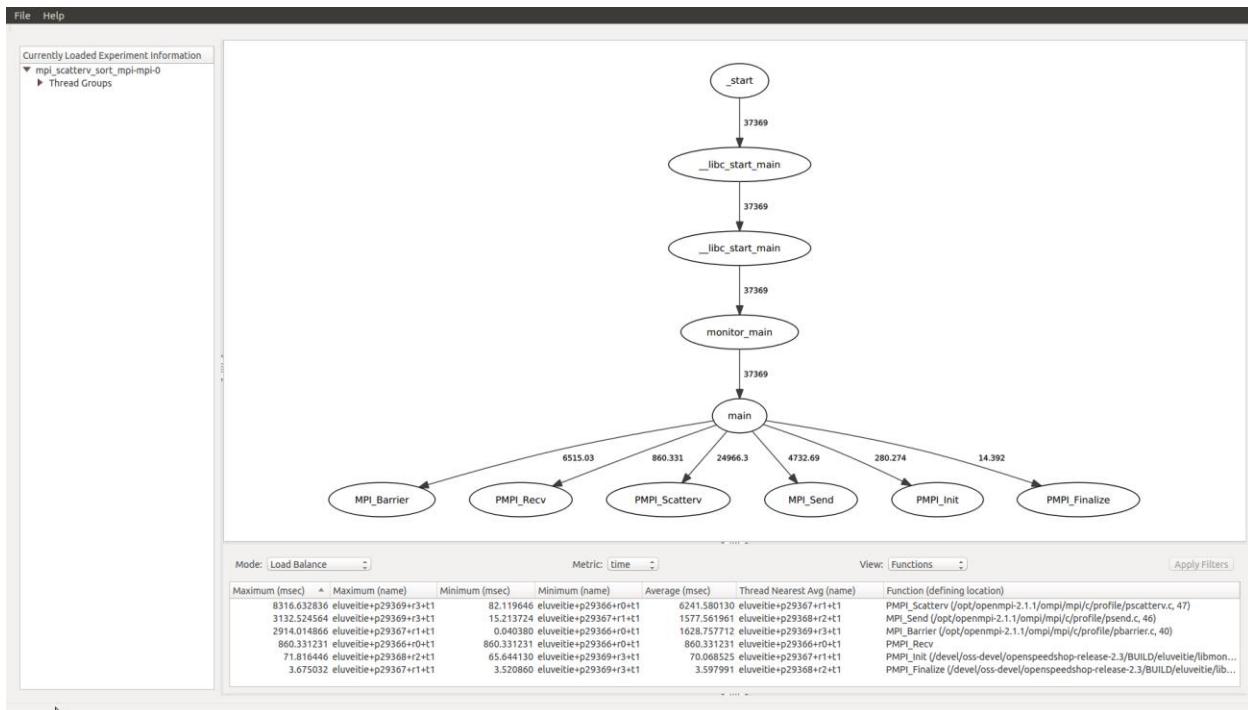


Figure 73 - load balance view (with calltree graph)

#### 9.1.4.12 Using the OSS 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 74).

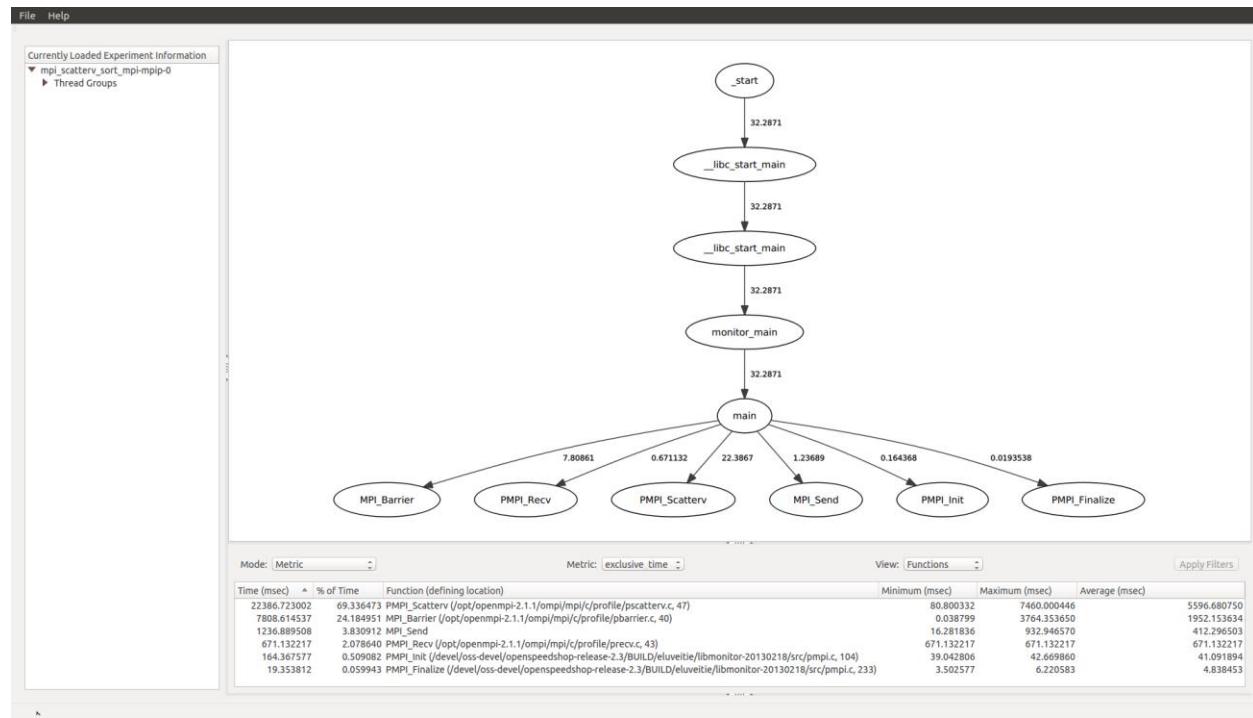


Figure 74 - mpip experiment default view (with calltree graph)

Additional views of possible interest to help point out imbalance of processing between processes or ranks are the Compare By Process (ref. Figure 75); Compare By Rank (ref. Figure 76) views; and Compare (ref. Figure 77) views. These are generated by selecting the “Compare By Process”, “Compare By Rank” or “Compare” option respectively 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 78).

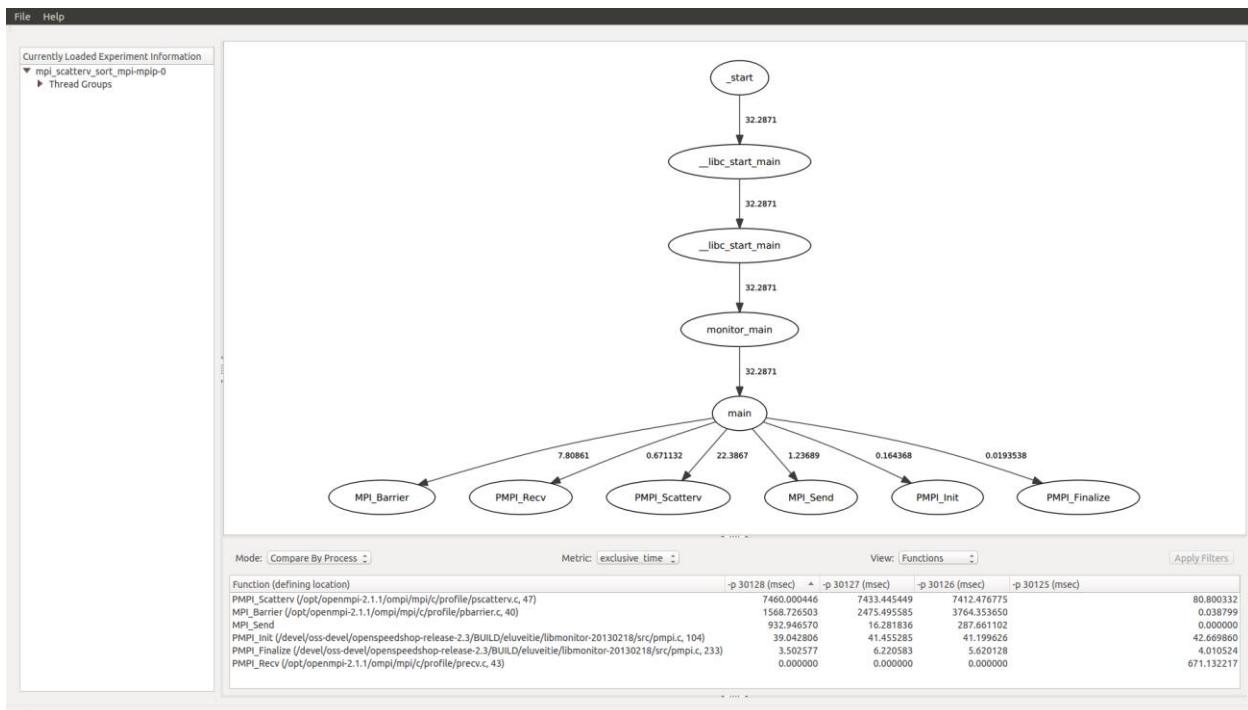


Figure 75 - compare by process view (with calltree graph)

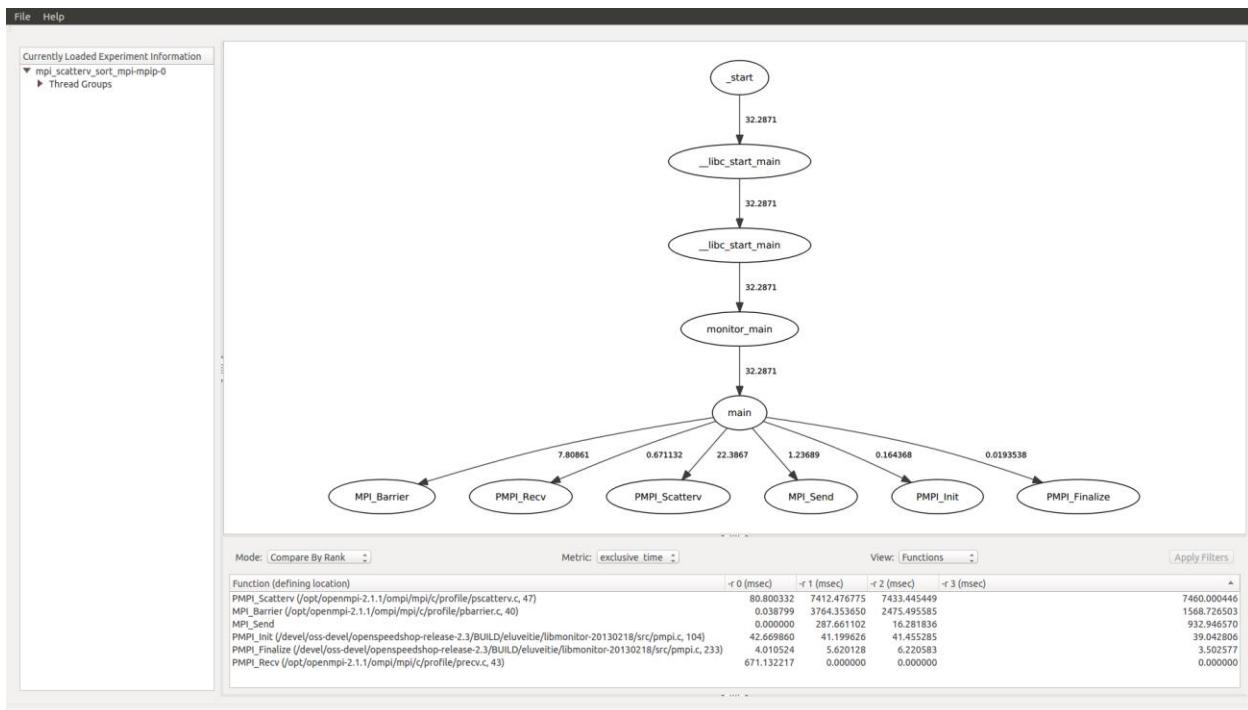


Figure 76 - compare by rank view (with calltree graph)

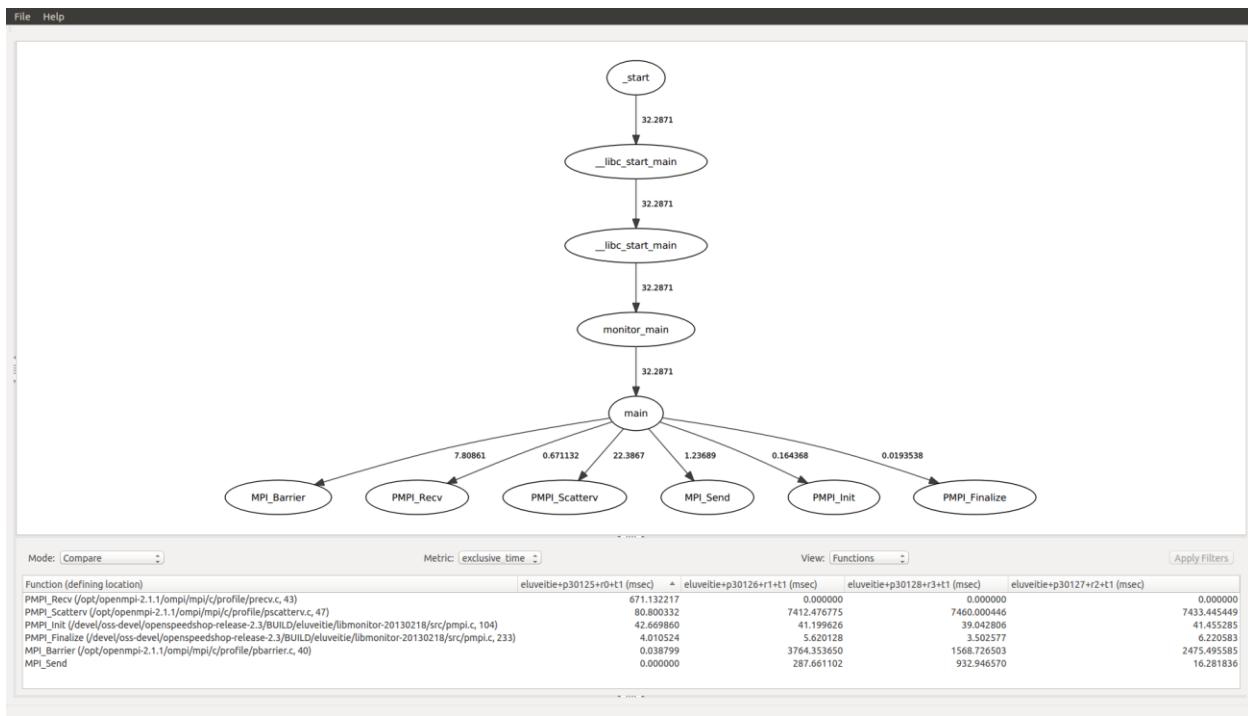


Figure 77 - compare view (with calltree graph)

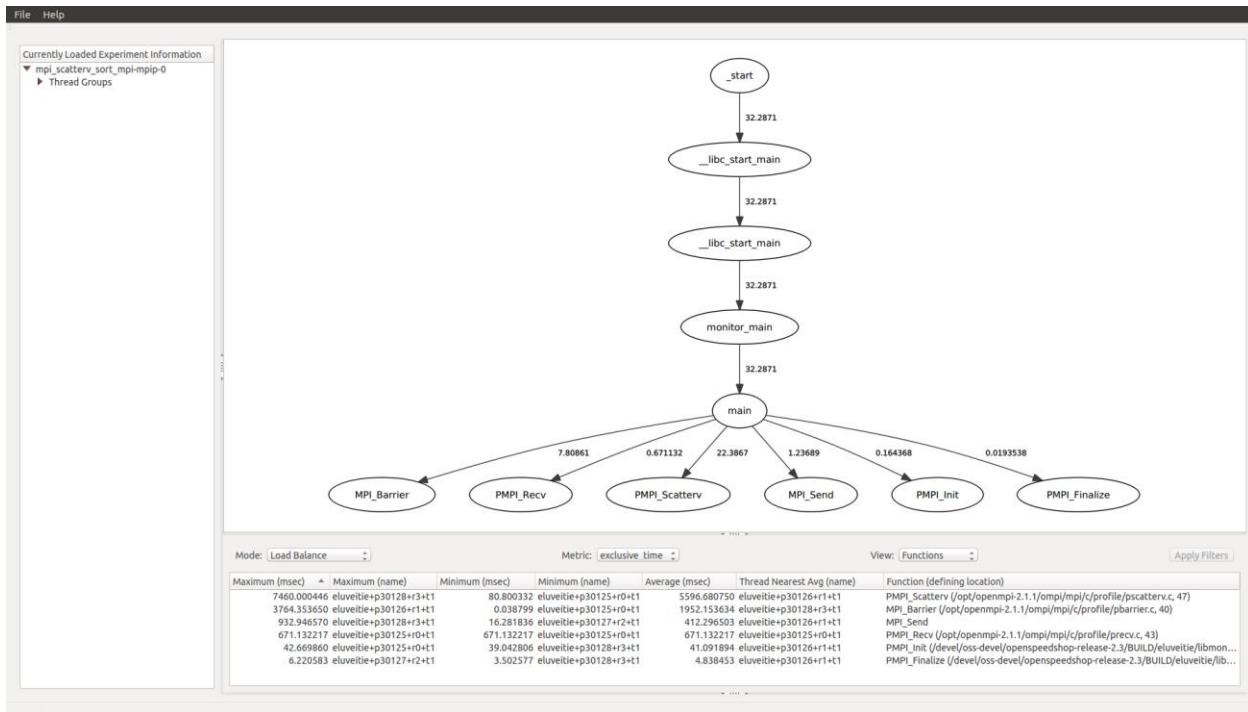


Figure 78 - load balance view (with calltree graph)

#### 9.1.4.12 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 79).

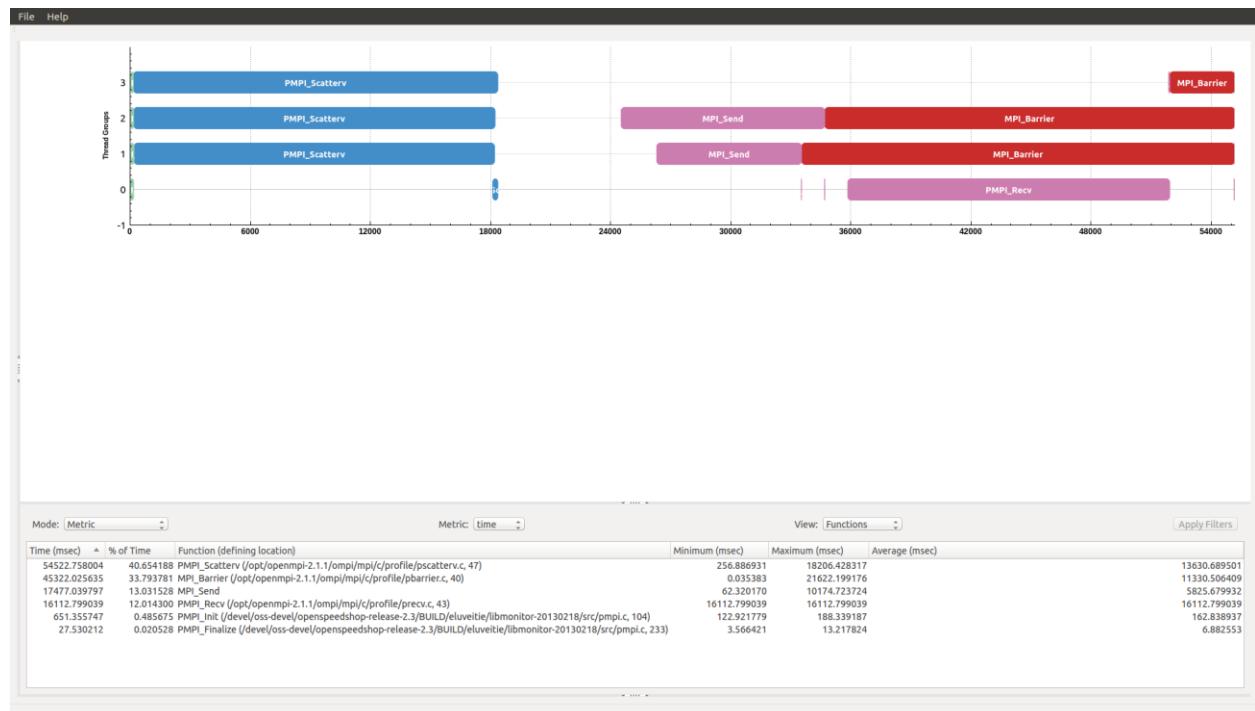


Figure 79 - mpit experiment default view (unbalanced)

For the “mpit” experiment case-study discussed here, the MPI application being profiled by Open|SpeedShop implements a simple parallel merge sort algorithm. The root rank (rank 0) divides the work up into a number of parallel processing units equal to the number of MPI processes specified during the MPI execution (i.e. “mpirun –np 4 ...”). The root process provides each rank a subset of the vector (subvector) to be sorted using the MPI\_Scatterv call. The subvector is sorted and then sent back to the root process via MPI\_Send. The root process waits for the sorted subvector using MPI\_Recv. As each sorted subvector is received by the root process it is merged into the final sorted vector. The relevant source-code snippet for the MPI parallel merge sort program can be seen in Figure 83.

The first series of screenshots (Figures 79 – 81) show results from a version of the MPI parallel merge sort application that generated subvectors of vastly unequal sizes. Thus, these initial screenshots show a MPI load unbalance. A version of the MPI parallel merge sort application, which had generated equal sized subvectors, will be compared to show MPI load balancing. Consequently, performance was improved and reduced the overall run time of the application. The output generated during the executions of the unbalanced and balanced MPI parallel

merge sort application can be seen in Figure 82. The unbalanced MPI program output is on the left and the balanced output is on the right.

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 “Compare By Rank” option from the “Mode” combo-box to view the load balance between ranks (ref. Figure 80). Note that the MPI\_Send calls (in ranks 1, 2, and 3) and MPI\_Barrier calls (rank 3 compared to ranks 1 and 2) show some big variation in values which may indicate there is a load unbalance in the application especially since there was an expectation that each parallel processing unit (rank) is doing equal work and should complete in roughly the same time.

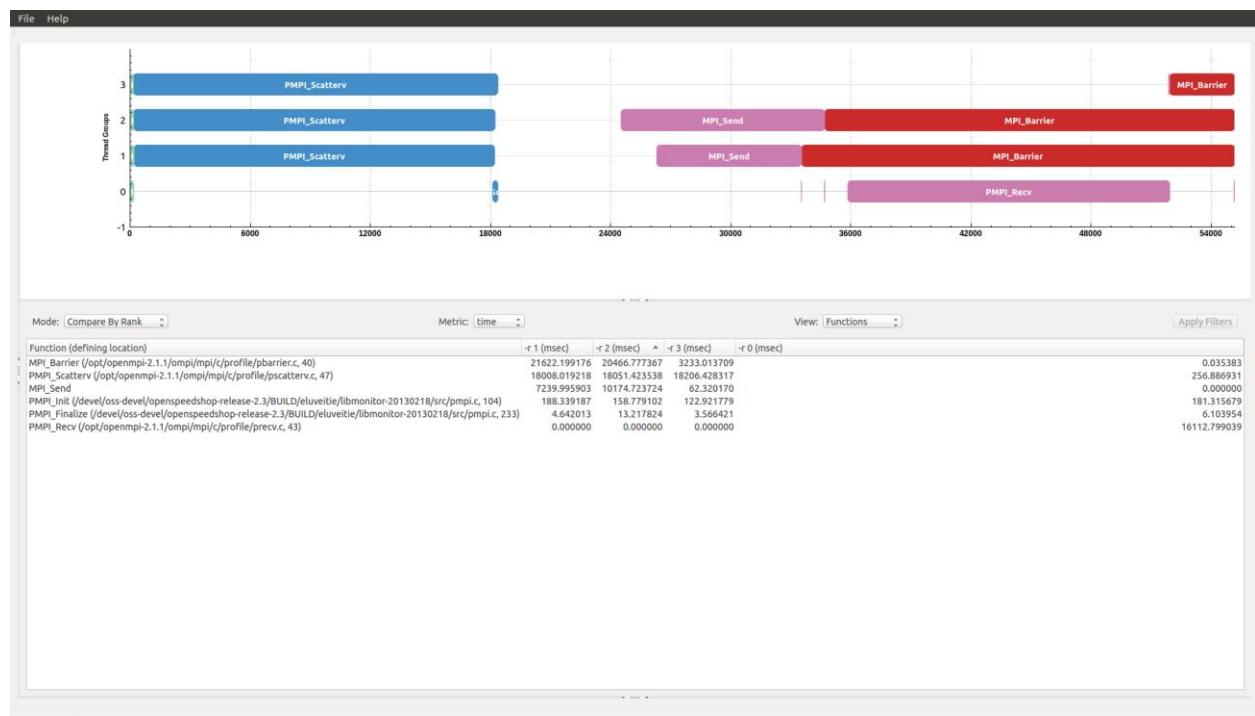


Figure 80 - mpit experiment compare by rank view (unbalanced)

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 81). In Figure 81 it can be noted that the message sizes are different for the MPI\_Send and MPI\_Recv calls (observe the sizes listed under the “Message Size” column).

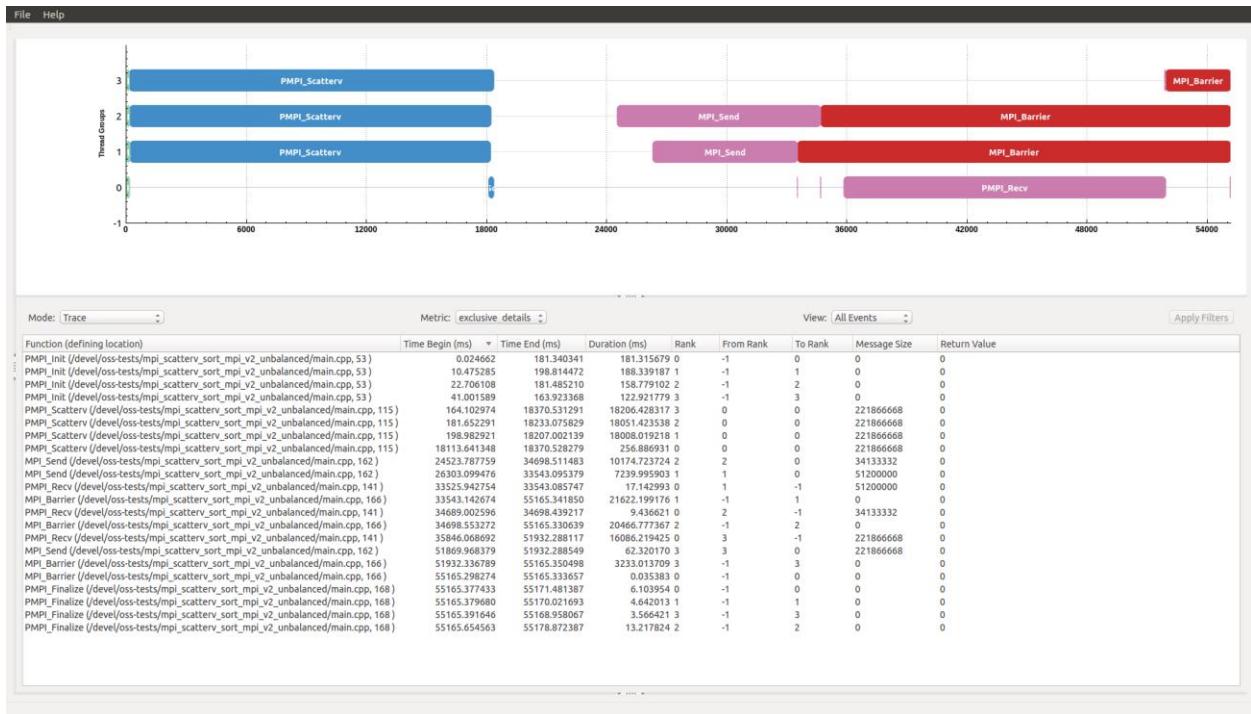


Figure 81 - MPI event list in Metric Table View (unbalanced)

Let's resolve the load balance issue, rerun the Open|SpeedShop experiment and load the resulting “mpit” experiment database. The output from the execution of the unbalanced and balanced versions of the application are shown in Figure 82. The send counts (“sendcounts”) for the balanced execution (right-side of Figure 82) are equal whereas the send counts for the unbalanced execution (left-side) are unequal. The pseudo source-code for both the unbalanced and balanced versions of the application are shown in Figure 83. The source-code to compute the send counts (“sendcounts”) and buffer offsets (“displs”) is not shown. The difference between the two versions are the values passed to the “sendcounts” and “displs” parameters of the MPI\_Scatterv call. The default view again appears showing the MPI event timeline and exclusive time metric values for the functions view (ref. Figure 84).

<pre> initialization of input vector finished! 0: sendcounts=25600000, displs=0 1: sendcounts=12800000, displs=25600000 2: sendcounts=8533333, displs=38400000 3: sendcounts=55466667, displs=46933333 Rank #0 Before MPI_Scatter Rank #1 After MPI_Scatter Rank #1 START SORTcd ../../. Rank #2 After MPI_Scatter Rank #2 START SORT Rank #0 After MPI_Scatter Rank #0 START SORT Rank #3 After MPI_Scatter Rank #3 START SORT Rank #2 FINISH SORT Rank #2 Before MPI_Send Rank #1 FINISH SORT Rank #1 Before MPI_Send Rank #0 FINISH SORT Rank #0 Before MPI_Recv Rank #1 After MPI_Send Rank #0 After MPI_Recv Rank #0 Before MPI_Recv Rank #2 After MPI_Send Rank #0 After MPI_Recv Rank #3 After MPI_Send RESULT = PASSED  real    0m59.191s user    1m55.216s sys     0m1.220s </pre>	<pre> initialization of input vector finished! 0: sendcounts=25600000, displs=0 1: sendcounts=25600000, displs=25600000 2: sendcounts=25600000, displs=51200000 3: sendcounts=25600000, displs=76800000 Rank #0 Before MPI_Scatter Rank #1 After MPI_Scatter Rank #1 START SORT Rank #2 After MPI_Scatter Rank #2 START SORT Rank #3 After MPI_Scatter Rank #3 START SORT Rank #0 After MPI_Scatter Rank #0 START SORT Rank #1 FINISH SORT Rank #1 Before MPI_Send Rank #2 FINISH SORT Rank #2 Before MPI_Send Rank #3 FINISH SORT Rank #3 Before MPI_Send Rank #0 FINISH SORT Rank #0 Before MPI_Recv Rank #1 After MPI_Send Rank #0 After MPI_Recv Rank #0 Before MPI_Recv Rank #2 After MPI_Send Rank #0 After MPI_Recv Rank #3 After MPI_Send RESULT = PASSED  real    0m44.041s user    1m31.788s sys     0m0.860s </pre>
---	--

Figure 82 - parallel merge sort execution output

```

100 float* h_x = NULL;
101
102 if ( 0 == world_rank ) {
103     h_x = (float *) malloc( N * sizeof(float) );
104     // fill the initial vector with random numbers
105 }
106
107 // generate arrays of send counts and offsets for MPI_Scatterv call
108
109 int* sendcounts = (int *) malloc( world_size * sizeof(int) );
110 int* displs = (int *) malloc( world_size * sizeof(int) );
111
112 int num_elements_per_proc; // = maximum value in sendcounts array
113
114 // For each process, create a buffer that will hold a subset of the entire array
115 float* proc_h_x = (float *) malloc( sizeof(float) * num_elements_per_proc );
116
117 // Scatter the random numbers from the root process to all processes in the MPI world
118 MPI_Scatterv( h_x, sendcounts, displs, MPI_FLOAT, proc_h_x, num_elements_per_proc, MPI_FLOAT, 0, MPI_COMM_WORLD );
119
120 std::sort( proc_h_x, proc_h_x+sendcounts[world_rank] );
121
122 if ( 0 == world_rank ) {
123     // NOTE: reuse original unsorted vector for receiving sorted subvectors and inplace merge sort
124     // copy local sorted vector into reused buffer
125     memcpy( h_x, proc_h_x, sizeof(float) * sendcounts[0] );
126     // free unsorted vector
127     free( proc_h_x );
128
129     // for each non-root rank
130     for ( int i=1; i<world_size; ++i ) {
131         // set number of elements expected from the current rank being processed
132         const int& num_elements = sendcounts[ i ];
133
134         // At the root process receive results from each child processes into the proper section of the vector being merged
135         MPI_Recv( h_x+displs[i], num_elements, MPI_FLOAT, i, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE );
136
137         // in-place merge
138         std::inplace_merge( h_x, h_x+displs[i], h_x+displs[i]+num_elements );
139     }
140 }
141 else {
142     MPI_Send( proc_h_x, sendcounts[world_rank], MPI_FLOAT, 0, 0, MPI_COMM_WORLD );
143 }
144
145 MPI_Barrier( MPI_COMM_WORLD );
146 MPI_Finalize();

```

Figure 83 - parallel merge sort source-code (snippet)

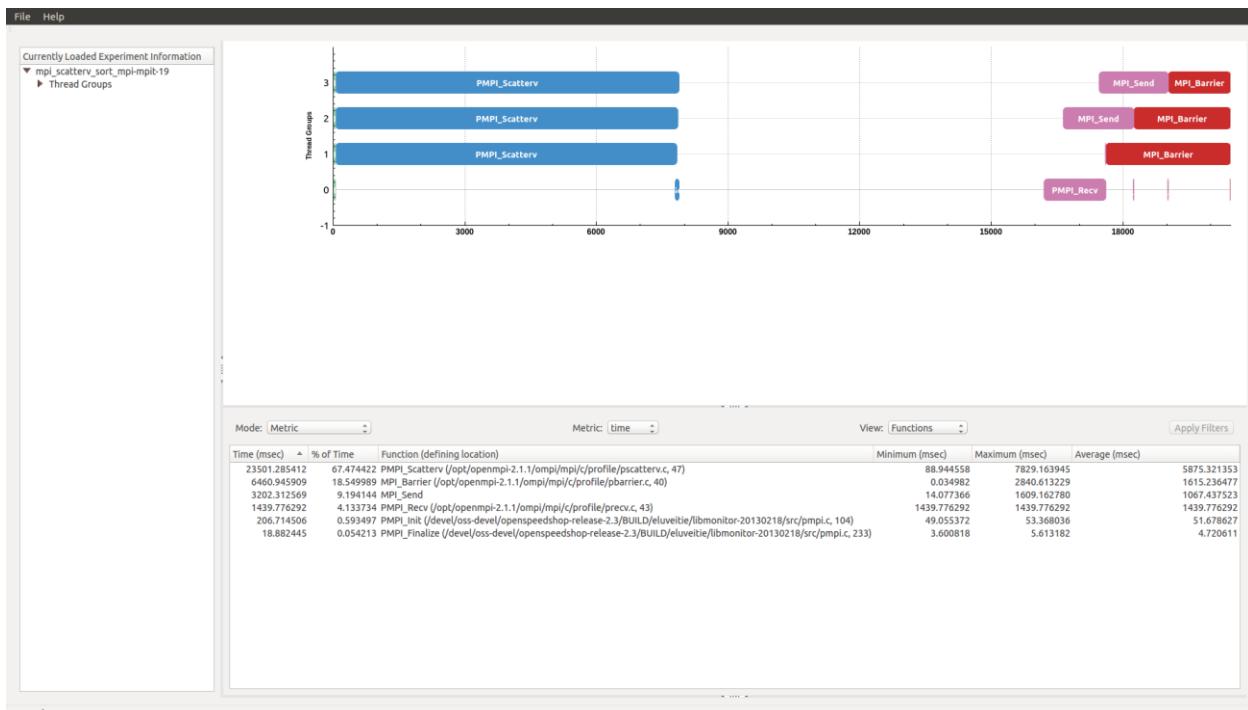


Figure 84 - mpit experiment default view (balanced)

Select “Compare By Rank” option from the “Mode” combo-box to view the load balance between ranks (ref. Figure 85). Note that the MPI\_Send and MPI\_Barrier calls, which showed large variation in values in Figure 80, now have values that are more equal and reduced in value compared to the maximum value in Figure 80. As seen in Figure 85, the MPI\_Scatterv calls have a time metric of roughly 7800 milliseconds which is vastly reduced from the durations of roughly 18000 milliseconds shown in Figure 80.

Also to be noted in Figure 85 for ranks 1 thru 3, the distance between the blue colored items in the MPI timeline graph (the MPI\_Scatterv calls) and the purple colored items (the MPI\_Send calls) are more consistent than as seen in Figure 79-81. This is also an indication that load balance has been achieved. The empty space represents the area where the std::sort function is called on a particular subvector to be processed by the MPI rank. For this example, the load balance was achieved by insuring the subvectors to be sorted are equal. It should be noted that one of the subvectors may be slightly different in size since this MPI application can deal with input vector sizes that are not equally divisible by the world rank size. Since the subvectors are equal (or very closely equal), the MPI\_Scatterv call duration has been reduced, the time to complete the sorting in parallel is much closer to the theoretical (time to sort the input vector sequentially divided by the world rank size) and the time to send the sorted subvectors back to the root rank is consistent as well.

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 86). Now notice that the message sizes for the MPI\_Send and MPI\_Recv calls are all equal.

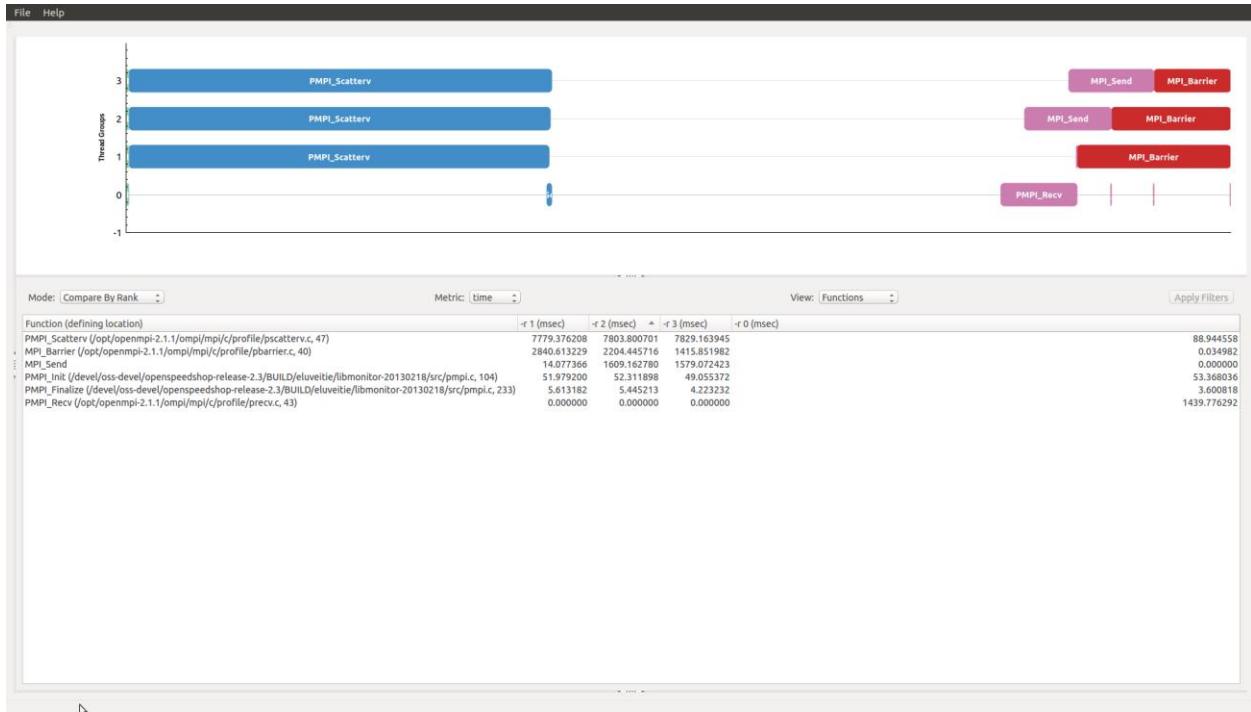


Figure 85 - mpit experiment compare by rank view (balanced)

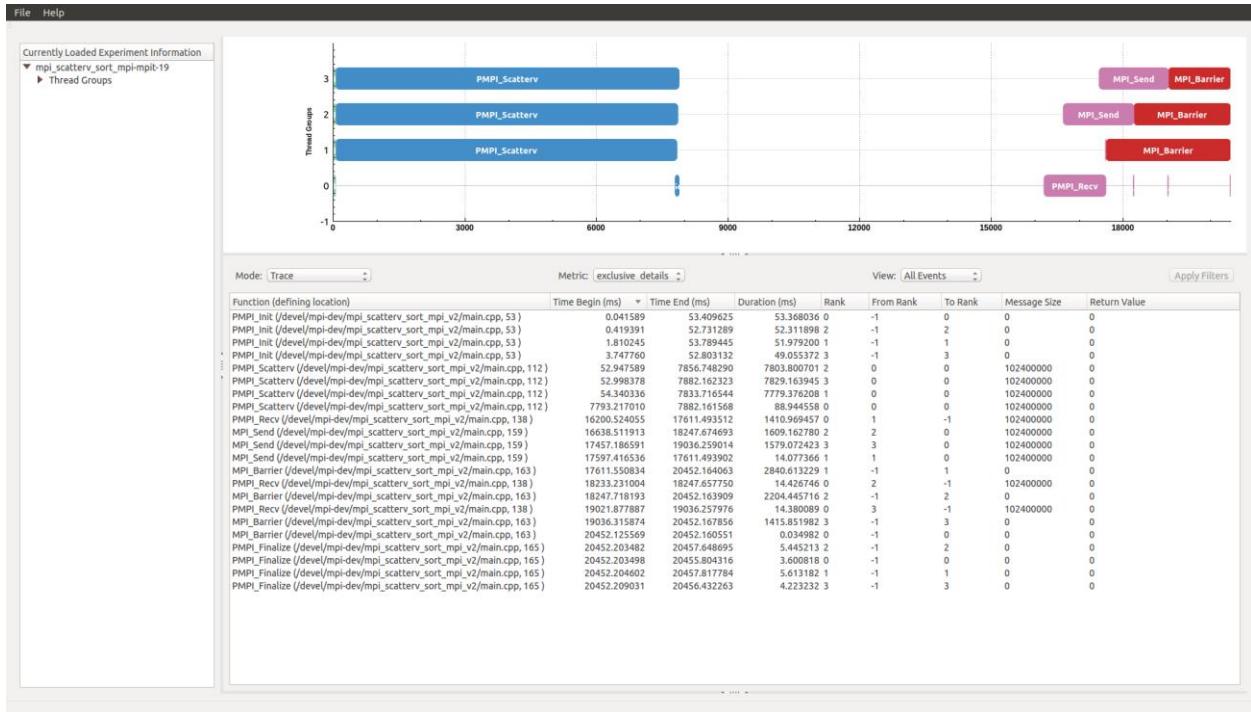


Figure 86 - MPI event list in Metric Table View (balanced)

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 87) and (ref. Figure 88).

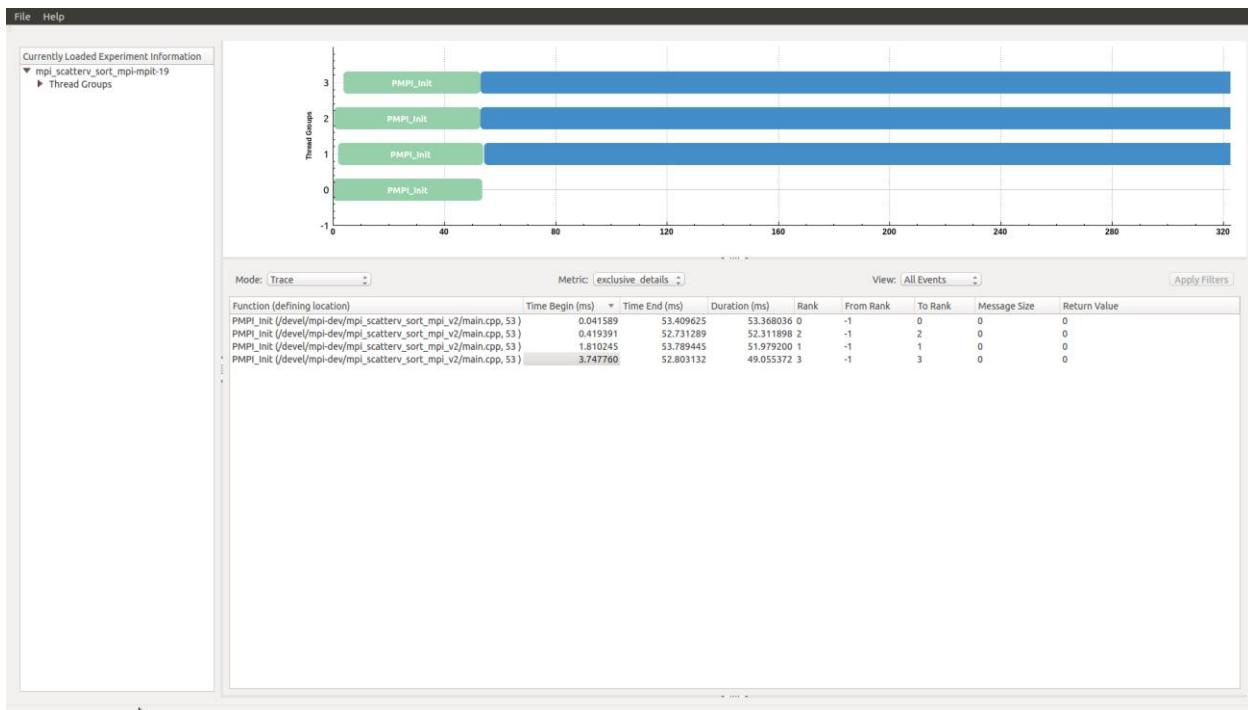


Figure 87 - MPI event list in Metric Table View (filtered to graph range – from experiment origin)

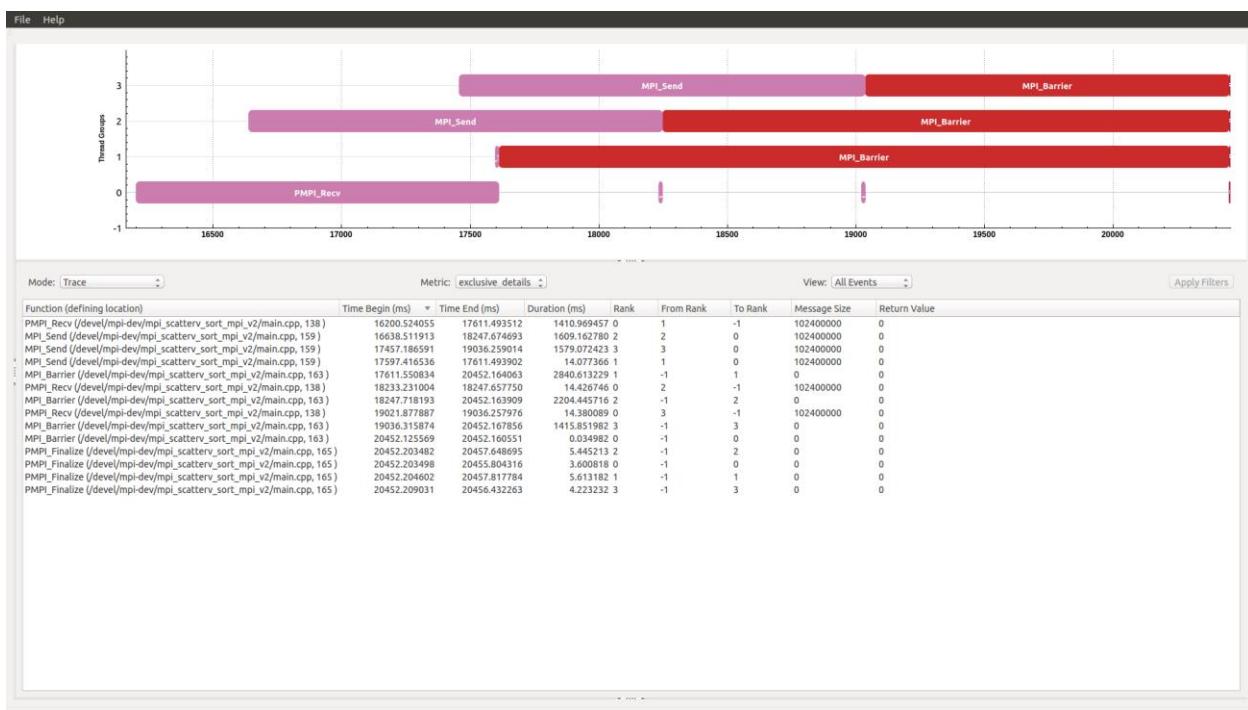


Figure 88 - MPI event list in Metric Table View (filtered to graph range – at experiment end range)

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 89). In addition, a dashed bounding rectangle is also drawn to help locate the event within a crowded event timeline (ref. Figure 90).

The dashed bounding rectangle remains visible for 10 seconds during which time the graph may be zoomed into the area being highlighted (ref. Figure 91). For the “Time Begin (ms)” item selected in Figure 90, 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.

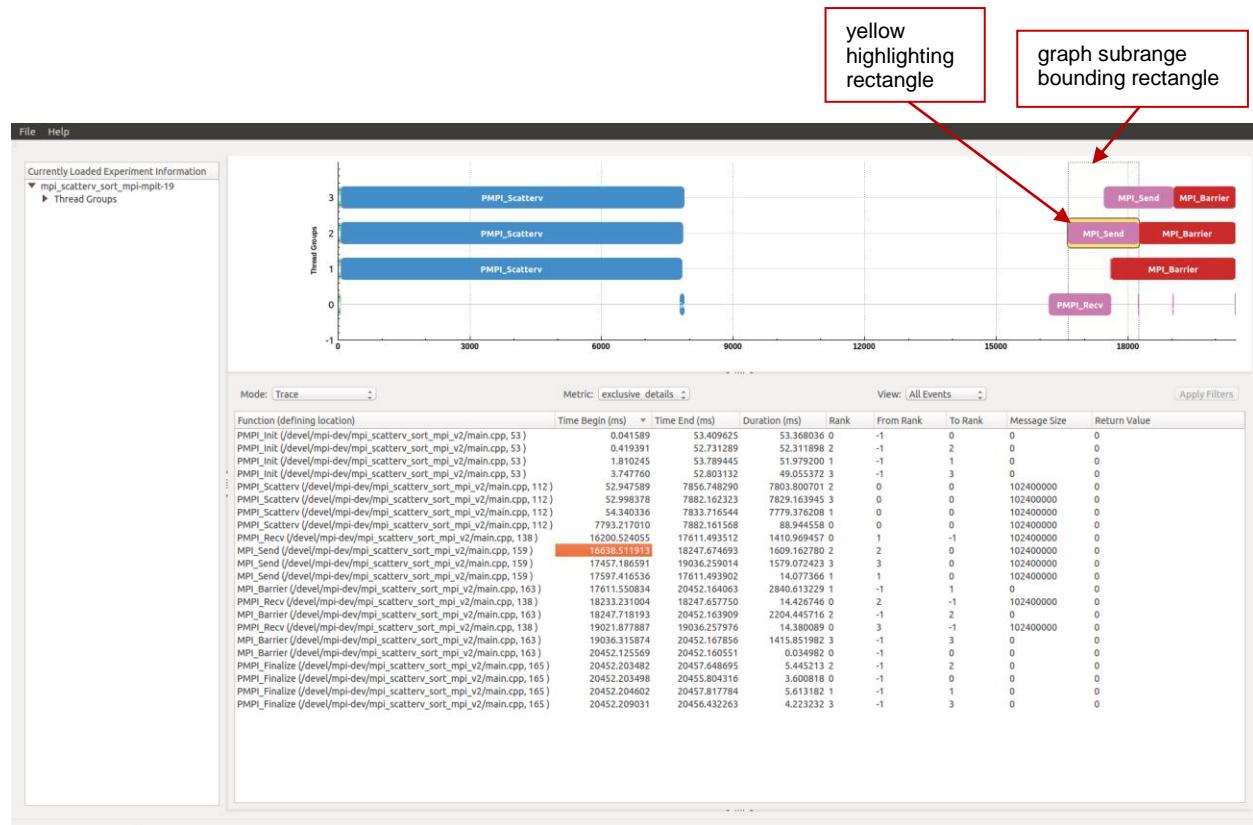


Figure 89 - locating MPI event in graph timeline

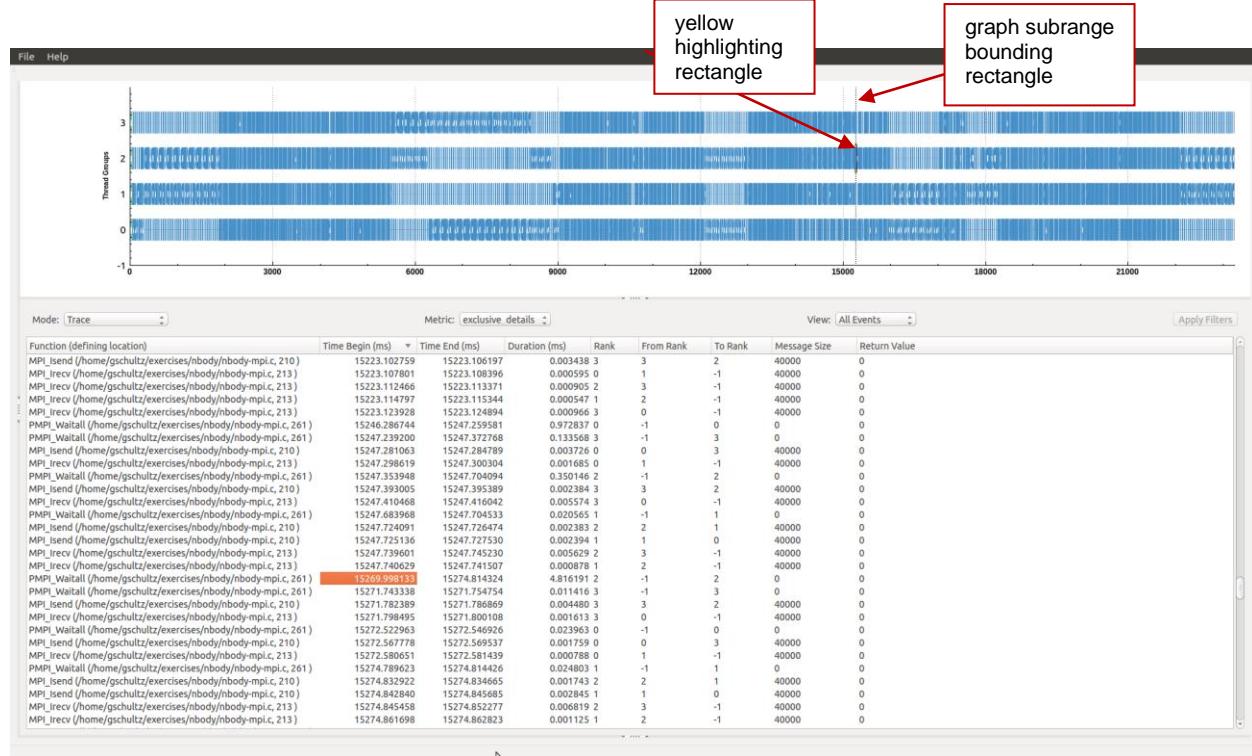


Figure 90 - locating MPI event in a crowded MPI event timeline

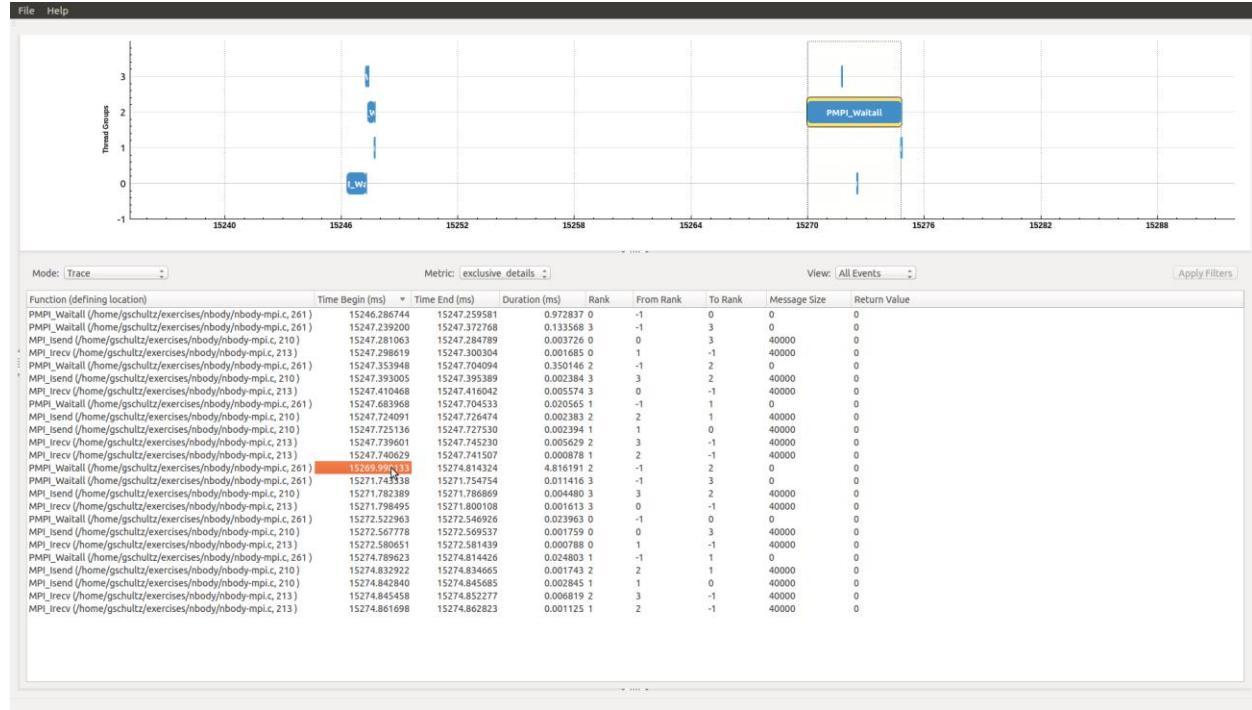


Figure 91 - Using highlighting cues to zoom into selected event

#### 9.1.4.13 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 92).

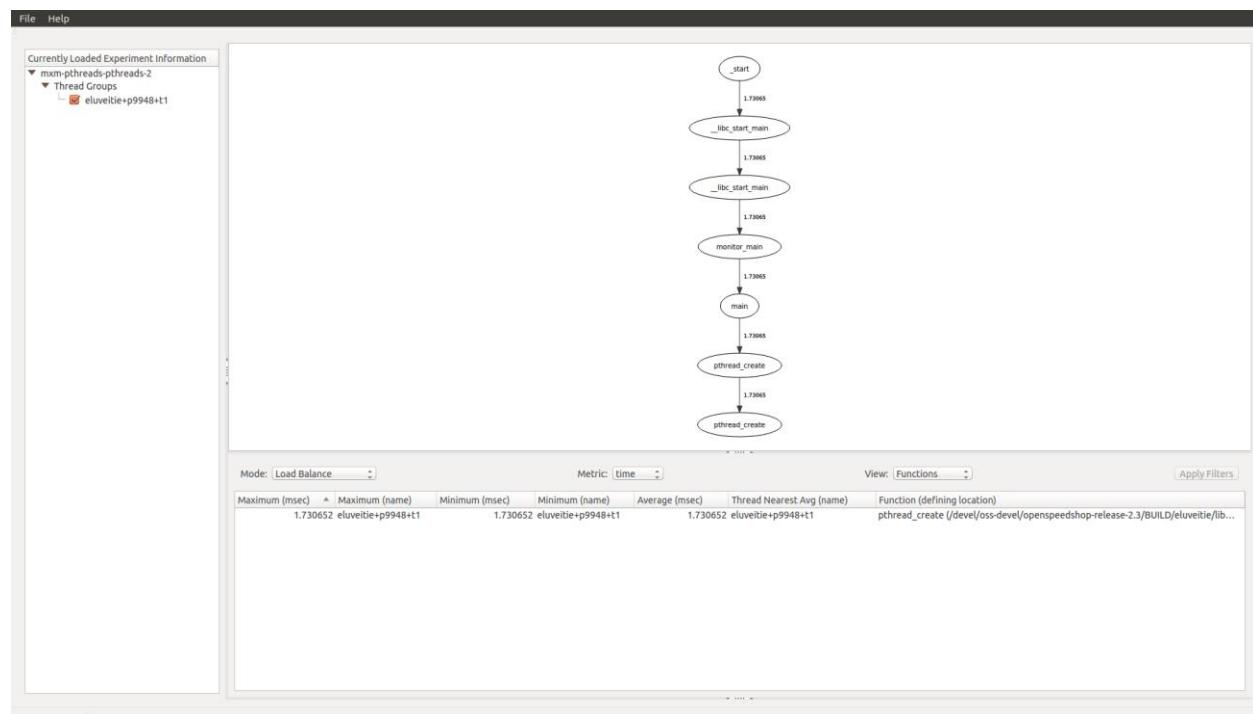


Figure 92 - pthreads experiment default view (with calltree graph)

#### 9.1.4.14 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 93. 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 93 and Figure 94 for a zoomed in view of the splitter handle between the Metric Plot and Metric Table Views).

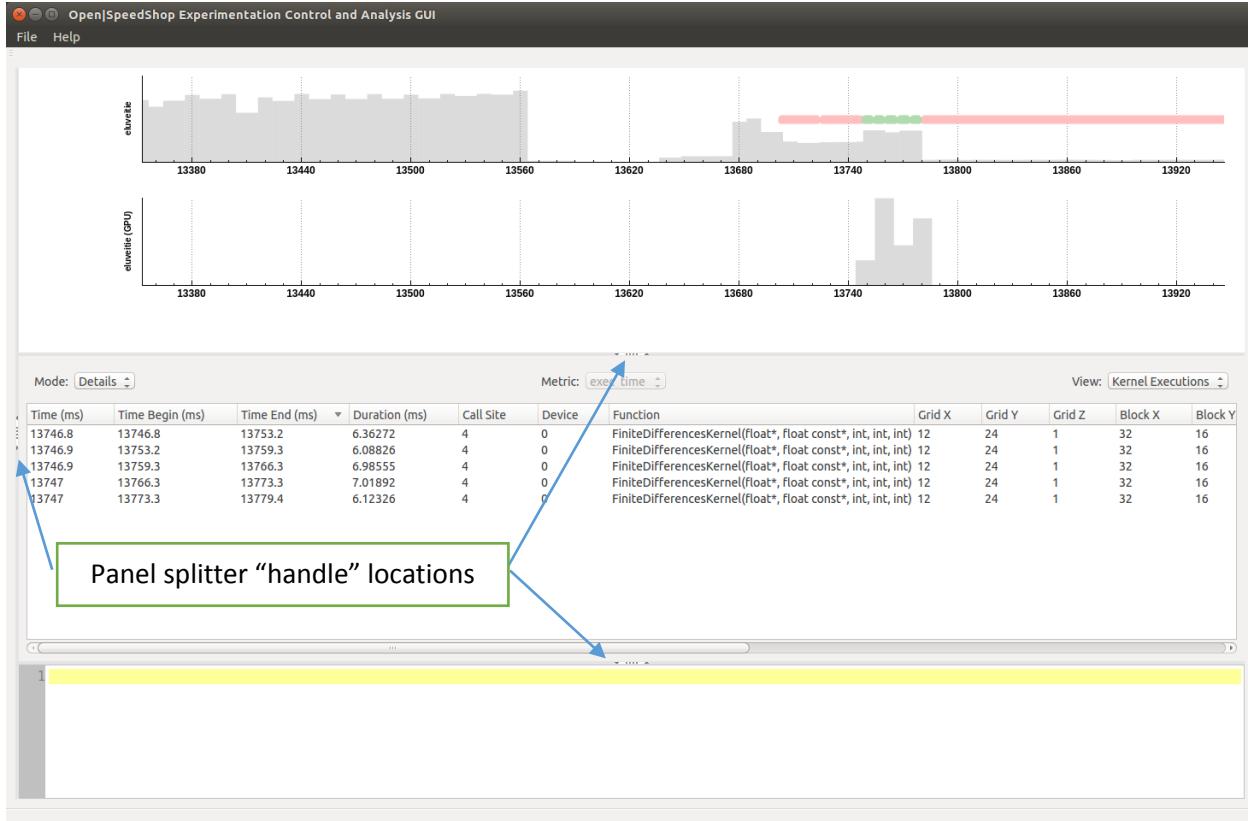


Figure 93 - Default View for the GEMM Experiment



Figure 94 - Zoomed View of Panel Splitter Handles

For the screenshot shown in Figure 93 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 rational 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 95 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 96 and 97).

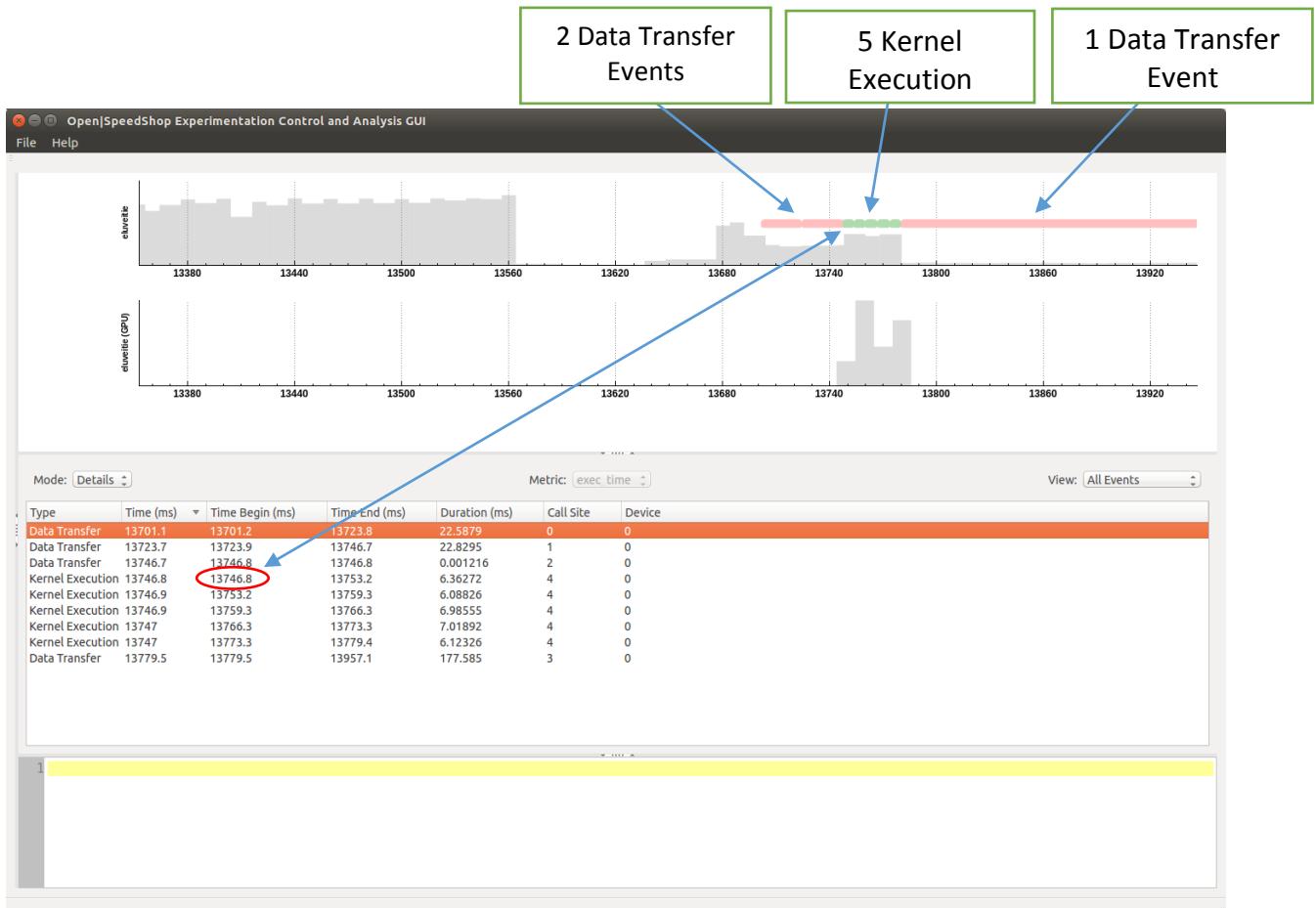


Figure 95 - CUDA Events in Graph Timeline and Details Mode View

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 96 - 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	6.9855	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 97 - Kernel Execution Details View

Another CUDA example will be discussed for a CUDA program which executes several different implementations of matrix multiplication using various 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>.

The following shows the output from running the “osscuda” convenience script causing the application to be profiled by the Open|SpeedShop CUDA collector which stores collected data in the Open|SpeedShop database.

```
$ 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 Open|SpeedShop 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 Open|SpeedShop 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   (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 -vxfex -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 series of screenshots shown in Figures 98-101 show the view configuration to achieve the same performance metric results in the GUI as obtained using the CLI.

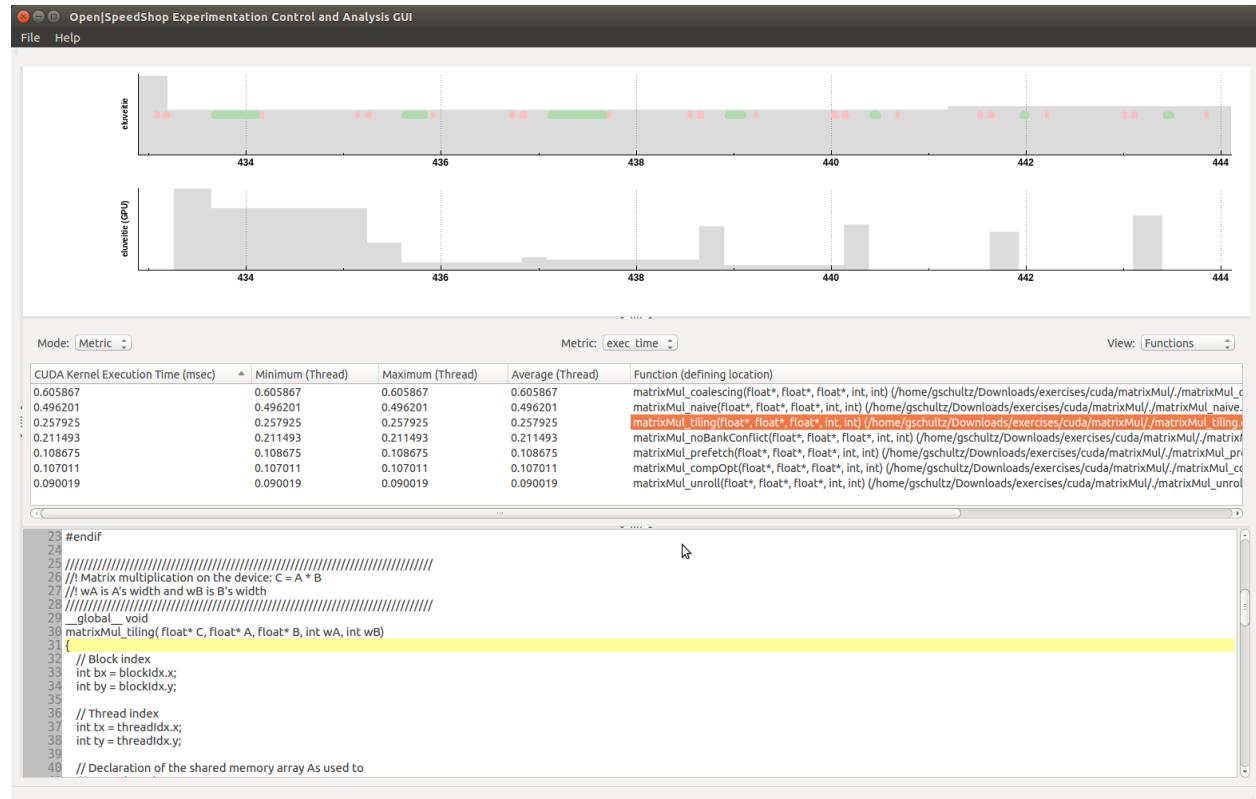


Figure 98 - "expview -vexec -mexclusive\_time,threadmin,threadmax,avg -I432.892:444.104"

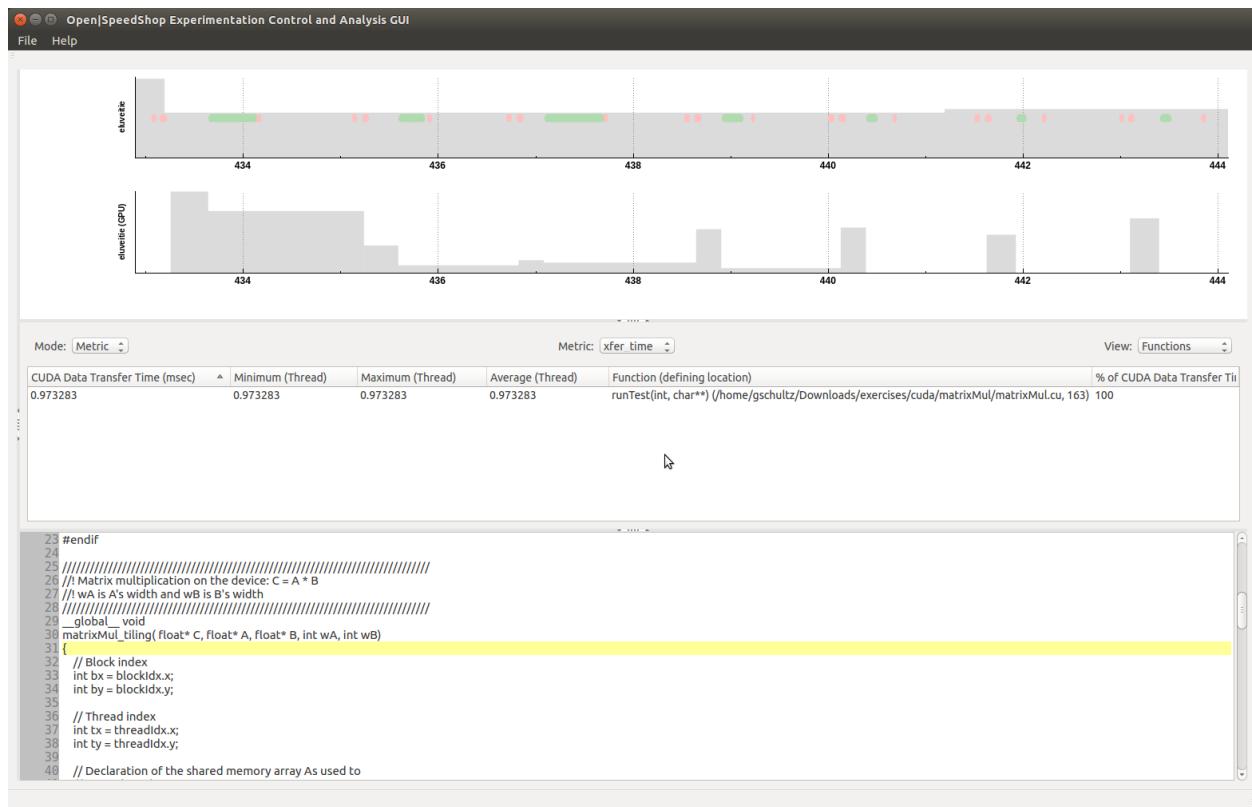


Figure 99 - "expview -vxfer -mexclusive\_time,threadmin,threadmax,avg -l432.892:444.104"

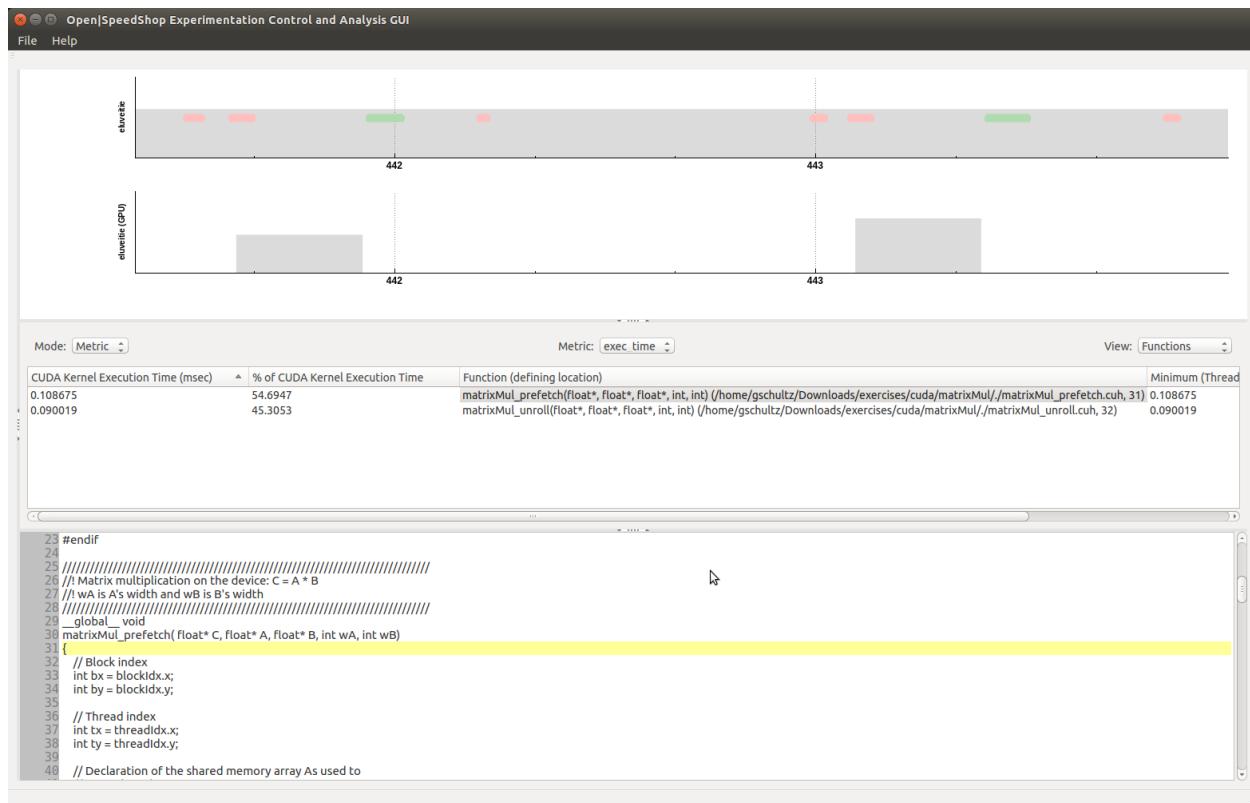


Figure 100 - "expview -vexec -mexclusive\_time,threadmin,threadmax,avg -l441.384:443.981"

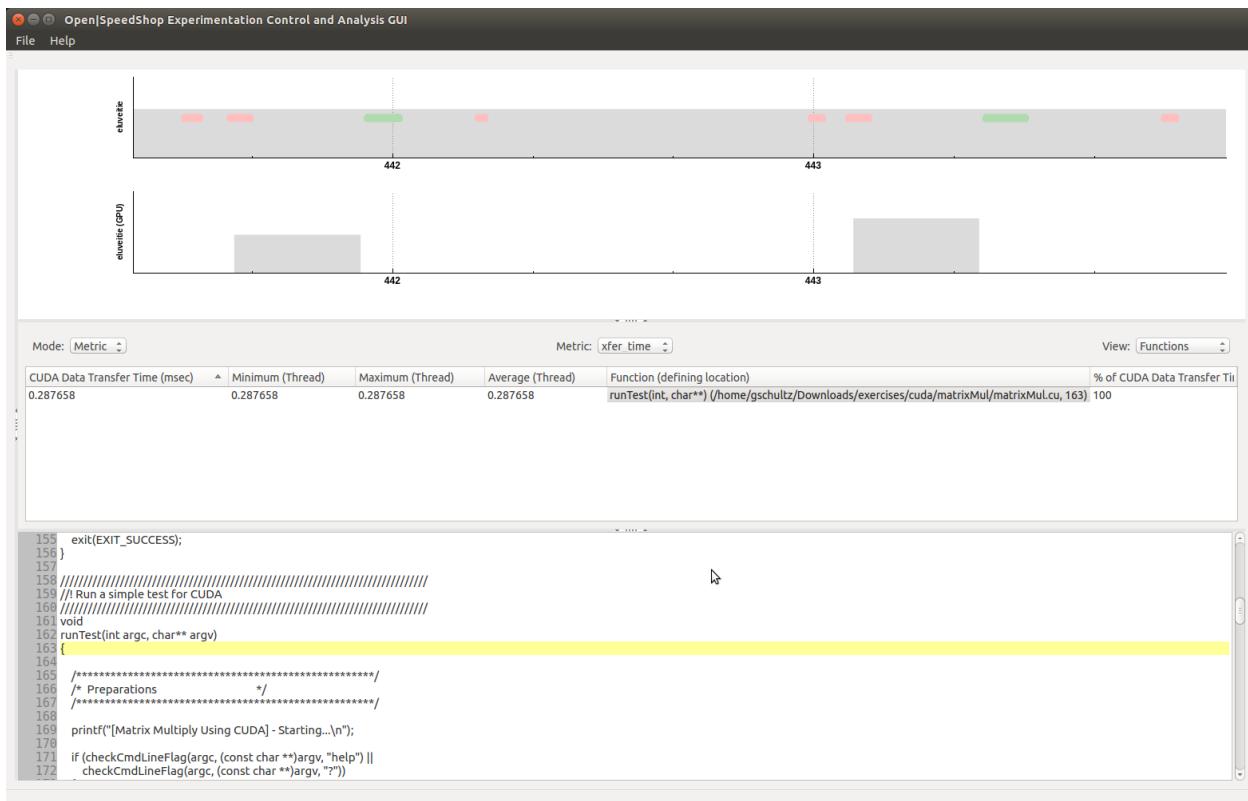


Figure 101 - "expview -vxfer -mexclusive\_time,threadmin,threadmax,avg -I441.384:443.981"

Each screenshot caption indicates the corresponding “expview” command in the Open|SpeedShop 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.

## Appendix A – Bibliography

- [1] *Open/SpeedShop Reference Guide, Version 2.3.1.* Contributions from Krell Institute, LANL, LLNL, SNL. November 11, 2017. URL [https://openspeedshop.org/wp-content/uploads/2017/11/OpenSpeedShop\\_Reference\\_Guide\\_v231\\_v8\\_standardsize.pdf](https://openspeedshop.org/wp-content/uploads/2017/11/OpenSpeedShop_Reference_Guide_v231_v8_standardsize.pdf).
- [2] Drepper, Ulrich. *What Every Programmer Should Know About Memory.* November 21, 2007. URL <https://www.akkadia.org/drepper/cpumemory.pdf>.