CodeAnalyst User's Manual

Publication: 26833

Revision 1.0

Publication date : March 4, 2010

Disclaimers

Trademarks

Table of Contents

List of Figures

Chapter 1. Introduction

1.1. Overview

AMD CodeAnalyst is a suite of tools to assist performance analysis and tuning. Chapter 2, *Features* provides a summary of CodeAnalyst features and concepts. It is essential reading for all CodeAnalyst users.

The overview is followed by the Chapter 8, *Tutorial*. The tutorial provides step-by-step directions for using CodeAnalyst. In order to get an overall impression of the CodeAnalyst workflow, please read following sections of these tutorials:

- Section 8.2, “Tutorial - Prepare Application”
- Section 8.3, “Tutorial - Creating a CodeAnalyst Project”
- Section 8.4, “Tutorial - Analysis with Time-Based Sampling Profile”
- Section 8.5, “Tutorial - Analysis with Event-Based Sampling Profile”
- Section 8.6, “Tutorial - Analysis with Instruction-Based Sampling Profile”

Following sections describe different areas of the CodeAnalyst configurations and workflow in detail.

- Chapter 3, *Types of Analysis*
- Chapter 4, *Configure Profile*
- Chapter 5, *Collecting Profile*
- Chapter 7, *View Configuration*
- Section 2.7, “Session Settings”
- Section 2.3, “CodeAnalyst Options”
- Section 2.5, “Importing Profile Data into CodeAnalyst”
- Section 2.6, “Exporting Profile Data from CodeAnalyst”

1.1.1. Preparing an Application for Profiling

AMD CodeAnalyst uses debug information produced by a compiler. Debug information is not required for CodeAnalyst profiling, but it is required for source-level annotation. Performance data can be collected for an application program that was compiled without debug information, but the results displayed by CodeAnalyst are less descriptive. For example, CodeAnalyst will not be able to display function names or source code. (Assembly code is displayed instead.) When compiling an application in release mode, the developer can still produce the debug information so that AMD CodeAnalyst can perform its analysis.

1.1.2. Compiling with the GNU GCC Compiler

When using GNU GCC to compile the application in general, specify the option **-g** to produce debugging information. Please refer to section **"Options for Debugging Your Program or GCC"** of the gcc Linux® manual page **(man gcc)** for more detail.

Chapter 2. Features

2.1. Overview of AMD CodeAnalyst

2.1.1. Program Performance Tuning

The program performance tuning cycle is an iterative process:

1. Measure program performance.
2. Analyze the results and identify program hot-spots.
3. Identify the cause for any performance issues in the hot-spots.
4. Change the program to remove performance issues.

AMD CodeAnalyst assists all four steps by collecting performance data, by analyzing and summarizing the performance data, and by presenting it graphically in many useful forms (tables, charts, etc.). CodeAnalyst directly associates performance information with software components such as processes, modules, functions and source lines. CodeAnalyst helps to identify the cause for a performance issue and where changes need to be made in the program.

The performance tuning cycle resembles the classic "scientific method" where a hypothesis (about performance) is made and then the hypothesis is tested through measurement. Measurement and analysis provide an objective basis for tuning decisions.

Performance analysis and tuning with CodeAnalyst consists of six steps:

1. Prepare the application for analysis by compiling with debug information turned on (an optional step).
2. Select the kind of data to be gathered by choosing one of several predefined profile configurations.
3. Configure run options such as the application program to be launched, the duration of data collection, etc.
4. Start and perform data collection.
5. Review and interpret the summarized results produced by CodeAnalyst.
6. Make changes to the program's algorithm and source code, recompile/link, and analyze again.

2.1.2. Types of Analysis

AMD CodeAnalyst is a suite of tools that help improve the performance of an application program or system. CodeAnalyst provides several different ways of collecting and analyzing performance data.

- **Time-based profiling (TBP)** shows where the application program or system is spending most of its time. This kind of analysis identifies hot-spots that are good candidates for tuning and optimization. After making changes to the code, time-based profiling can evaluate, measure, and assess improvements to performance. It can also verify that the modifications improved execution speed and calculate by how much. Please see Section 3.2, “Time-Based Profiling Analysis” for more detail.
- **Event-based profiling (EBP)** uses the performance monitoring hardware in AMD processors to investigate hot-spots. This kind of analysis identifies potential performance issues such as poor data access patterns that cause cache misses. An event-based profile can identify the reason for a performance issue as well as the code regions that may be performance culprits. Event-based profiling can test hypotheses about a performance issue to identify and resolve it. When multiple events are sampled, an event profile shows the proportion of one event to another. See Section 9.1, “Perfor-

mance Monitoring Events (PME)" for descriptions of the events supported by AMD processors. Please see Section 3.3, "Event-Based Profiling Analysis" for more detail.

- **Instruction-based sampling (IBS)** also uses the performance monitoring hardware. This kind of analysis identifies the likely cause of certain performance issues and associates those issues precisely to specific source lines and instructions. Please see Section 3.4, "Instruction-Based Sampling Analysis" for more detail.
- **Basic Block Analysis** statically analyzes the assembly instructions to identify basic blocks and aggregates data accordingly. Please see Section 3.5, "Basic Block Analysis" for more detail.
- **In-line Analysis** allows users to aggregate samples into either in-line functions or in-line instance. Please see Section 3.6, "In-Line Analysis" for more detail.
- **Session Diff** allows comparison of profiling sessions based on symbols. Please see Section 3.7, "Session Diff Analysis" for more detail.

Analysis usually begins with time-based profiling in order to find time-critical and time-consuming software components. Event-based profiling or instruction-based sampling is usually employed next in order to determine why a section of code is running more slowly than it should.

2.1.3. Flexible, System-Wide Data Collection

CodeAnalyst's data collection is system-wide, so performance data is collected about all software components that are executing on the system, not just the application program itself. CodeAnalyst collects data on application programs, dynamically loaded libraries, device drivers, and the operating system kernel. CodeAnalyst can be configured to monitor the system as a whole by **not** specifying an application program to be launched when data collection is started. Time-based profiling, event-based profiling, and instruction-based sampling collect data from multiple processors in a multiprocessor system. CodeAnalyst can also be used to analyze Java just-in-time (JIT) code.

2.1.4. Summarized Results with Drill-down

CodeAnalyst summarizes and displays performance information at several levels of granularity or "aggregation:"

- **Process**
- **Module**
- **Function**
- **Source line**
- **Instruction**

The CodeAnalyst graphical user interface organizes and displays information at each of these levels and provides drill-down. Thus, CodeAnalyst provides an overview of available performance data (by process or by module) followed by drill-down to functions within a module, to source lines within a function, or even the instructions that are associated with a line of source code.

2.1.5. Graphical User Interface

The CodeAnalyst graphical user interface (GUI) provides an interactive workspace for the collection and analysis of program and system performance data.

2.1.6. Projects and Sessions

The CodeAnalyst GUI uses a project- and session-oriented user interface. A project retains important settings to control a performance experiment such as the application program to launch and analyze,

settings that control data collection, etc. A project also organizes performance data into sessions. A CodeAnalyst session is created when performance data is collected through the GUI or when profile data is imported into the project. (The Oprofile command line utility is an alternative method for collecting data.) Session data is persistent and can be recalled at a later time. Sessions can be renamed and deleted.

2.1.7. Summary of GUI features

The CodeAnalyst GUI offers many features that make it easy to collect, view and analyze performance data. This subsection summarizes the main features offered by the CodeAnalyst GUI.

Organize and manage performance data in a CodeAnalyst project where a project consists of one or more sessions.

Configure and control program execution, and data collection such as:

- Specify data collection parameters (e.g., sampling interval, trigger events, trigger event count, inclusion of system/user mode samples, etc.)
- Manually start and stop data collection
- Delay data collection for a specified period of time to avoid taking measurements during a programs start up phase
- Stop data collection when the monitored program terminates or after a specified time (duration) has expired

Define important program properties and project options such as:

- Path to the executable binary image
- Path to program source code

Collect performance data using time-based profiling, event-based profiling, and instruction-based profiling where available.

Display performance information in different formats such as:

- Table
- Bar graph
- Annotated source and assembler code

Display performance information in different views such as:

- System Data and System Graph views to identify hottest code modules and CPUs
- Module Data and Module Graph views to drill down into a single module to identify hot procedures and code within a module
- Source and Assembly Views to see the time and event data associated with individual statements and instructions

Capture and save code produced by Java just-in-time (JIT) compilation.

Import performance data that was collected using the Oprofile command line utility.

Export data to allow post-processing using a spreadsheet program (or a user-written custom application).

2.1.8. Basic Steps for Analysis

The CodeAnalyst graphical user interface provides features to set up a performance experiment, run the experiment while collecting data, and display the results. The basic steps are:

1. Open an existing project or create a new project.
2. Set up basic run parameters like the program to launch, the working directory, etc.
3. Select a predefined profile (data collection) configuration.
4. Collect a time-based profile, event-based profile, or IBS-based profile as selected by the profile configuration.
5. View and explore the results.
6. Save the project and session data to review it later or to share it.

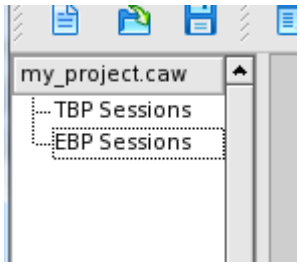
2.2. Exploring the Workspace and GUI

The Exploring the Workspace and GUI section serves as a visual guide to options and related screens. All options that have a separate information section contain links to that section.

2.2.1. Projects Panel

The Project Panel opens initially with no projects open. Projects consist of sessions created by using profile configurations and by importing performance data captured using the CodeAnalyst Command Line Utility program (OProfile). Kinds of sessions displayed are TBP (time-based sampling sessions), EBP (event-based sampling sessions), and IBS (Instruction-Based Sampling sessions).

Figure 2.1. Project Panel



2.2.2. Status Bar

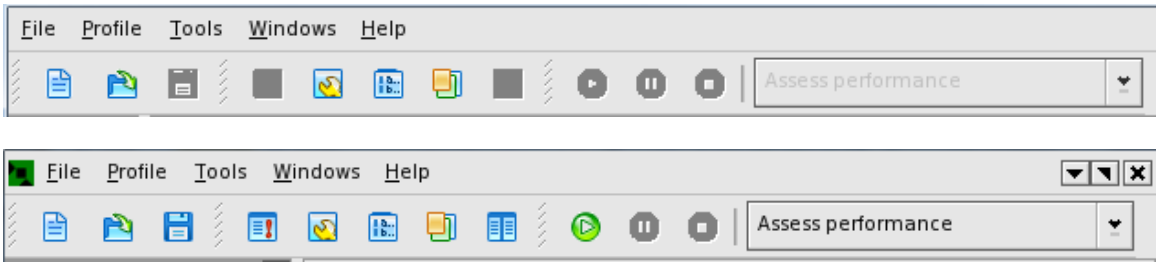
The status bar displays the current operation taking place. For example, while a profile is being collected, the Sampling Sessions Started status bar displays the amount of time left to run in percentages. The status bar displays some of the following examples:

Figure 2.2. Status bar

2.2.3. Toolbars

The CodeAnalyst tools consist of menu items and corresponding icons or drop-down lists. Most menu items and icons are not active until after a session is opened (or a profile session is running), as the following illustrations show.

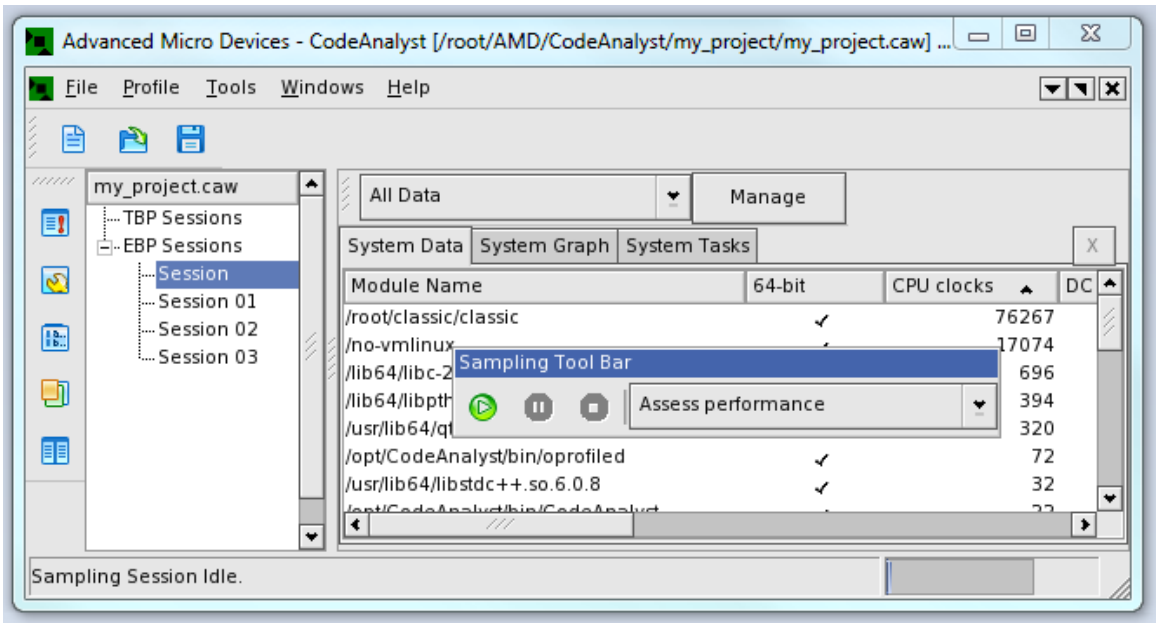
Figure 2.3. Toolbars - inactive and active



2.2.4. Floating and Docking Toolbar Groups

Toolbar groups have the ability to be floated or docked in the workspace. Drag and drop a toolbar group using the grip located to the left of the group. Drag into the work area until the border darkens, and release. The toolbar displays its group name in a header. Double-click on the toolbar name to automatically return it to the toolbar area. To replace in original position, drag and realign the group until the shape changes (elongates) and then release the mouse. Following are examples of toolbars being docked and floated.

Figure 2.4. Floating and docking toolbars



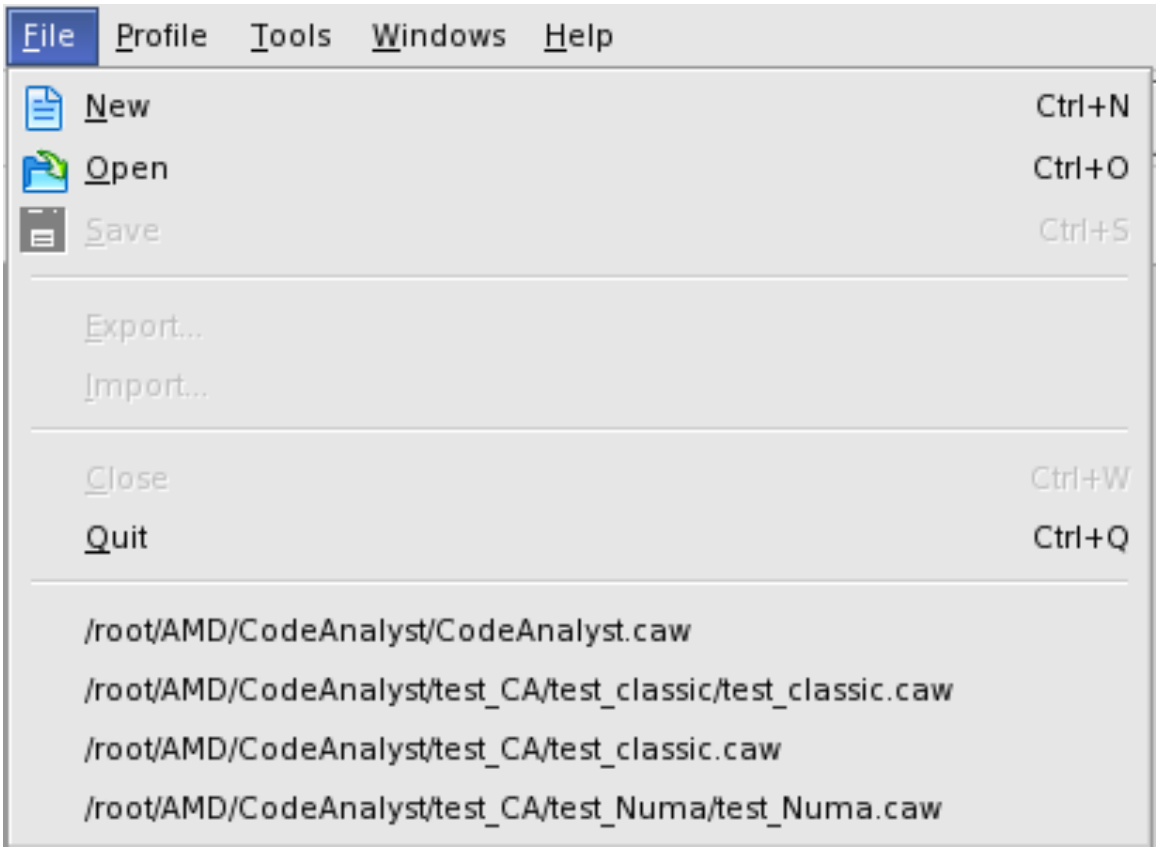
Any single icon or groups of icons that include the grip bar can be moved around the work area. This includes tools found only on specific tab windows.

2.2.5. Menus, Tools, and Icons

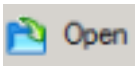
CodeAnalyst provides a menu bar and three toolbars. The toolbars can float in any area of the workspace.

The following sections give descriptions and definitions of menus, tools, and icons available for creating projects, profile actions, and creating configurations.

2.2.6. File Menu and File Icons



The commands available in the File menu are shown in the following table. These icons also appear as toolbar group icons that can float or be docked.

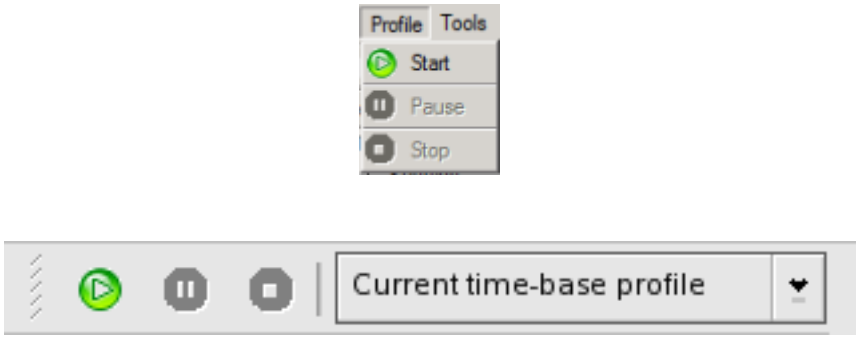
Menu Command and Icon	(Fast Key)	Description
	(Ctrl+N)	Opens a new project. This opens the Project Options dialog box.
	(Ctrl+O)	Opens an existing project. This opens a dialog box for navigating to CodeAnalyst workspace files. Recently opened projects are listed at the bottom of the File menu.
	(Ctrl+O)	Saves an open project.
Export System Data...		Exports the project data to a comma separated value (CSV) formatted file. This action opens the Save As dialog box of file type CSV.
Import...		Imports performance data files generated by Oprofile command-line tools.

Menu Command and Icon	(Fast Key)	Description
Close	(Ctrl+W)	Closes an open project.
Quit	(Ctrl+Q)	Closes the application.

2.2.7. Profile Menu and Toolbar Icon Group

The Profile menu and icons are used to start, pause, and stop data collection. Options are also part of the toolbar in the Profile toolbar group, which includes the profile configuration to be used for collecting performance data.

Figure 2.5. Profile menu and toolbar



The commands available for controlling data collection and for selecting the profile configuration are shown in the following table.

Icon	Description
Start 	Activates the sampling process in the same way as choosing Start from the Sampling menu. This icon is in the active state when a project is open and the sampling process has not yet started
Pause 	Suspends the sampling process in the same way as choosing Pause from the Sampling menu. This icon is red in the active state when a project is open and the sampling process is in progress
Stop 	Terminates the sampling process in the same way as choosing Stop from the Sampling menu. This icon is red in the active state when a project is open and the sampling process has started or has been paused.
[profile configurations]	(Second list) Provides a list of pre-defined profile configurations for data collection. The profile configuration determines the performance data to be collected (time-based, event-based, etc.). The list changes according to which mode is selected. See Chapter 4, *Configure Profile* for more details.

2.2.8. Tools Menu and Icons

The **Tools** menu contains icons for modifying project options for the current project and application-level CodeAnalyst options. These icons also appear as toolbar group icons that float or can be docked. As the last illustration shows, the Session Settings icon is not active unless a session is opened.

Figure 2.6. Tools menu and toolbar

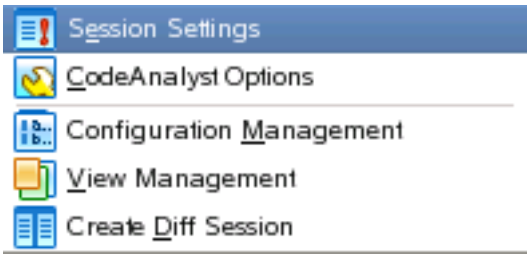
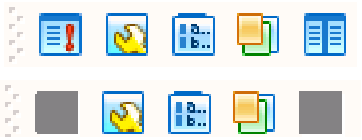


Figure 2.7. Active and Inactive Icons



The following table summarizes the **Tools** menu items and associated toolbar icons. Hovering the mouse over the icon displays the name as indicated in parentheses. Links under "Menu Command" go to the corresponding section for more information. The rollover text is included to assist with similar naming conventions used.

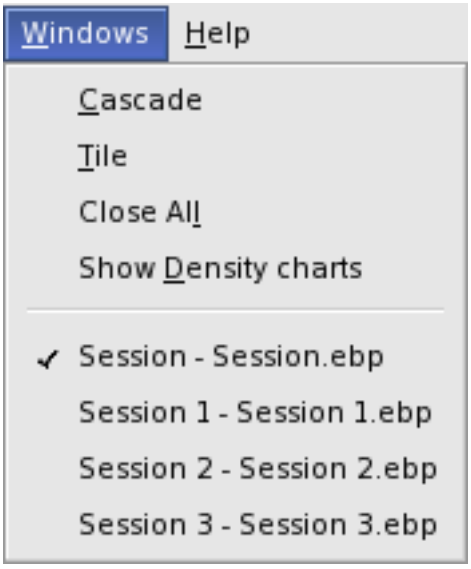
Menu Command / Icons	**Description**
Session Settings 	Use this icon to change the sessions settings (e.g., time-based and event-based). Please see Section 2.7, “Session Settings” for more detail.
CodeAnalyst Options... 	Opens the CodeAnalyst Options dialog box. All of the application level configuration options can be changed. Please see Section 2.3, “CodeAnalyst Options” for more detail.
Configuration Management 	Profile configurations determine how performance data is collected. Profiles configurations can be defined by the user or predefined configurations can be selected. For more information, see Chapter 4, *Configure Profile*.
View Management 	A "view" consists of a set of event data and computed performance measurements displayed together in a table or a graph. Use View Management to open the View Configuration dialog box and to change the contents displayed in the view. Exchange content from the Available Data list and the Column shown list. Items in the Column Shown list appear in the View Management window. Some views have prerequisites that must be met before they can be selected from the available data list. For more information, see View Configurations. The Manage button opens the view management dialog for the currently selected view. Please see Section 7.3, “View Management” for more detail.
Diff Session	Opens the Diff Session Dialog where users can choose any two sessions in the project to differentiate (diff) data in sessions using the AMD Dif-

Menu Command / Icons	Description
	fAnalyst tool. For detailed instructions using the tool, refer to the included document, *AMD DifffAnanlyst for Linux*, publication# 45919.

2.2.9. Windows Menu

The Windows menu controls the display attributes of the Data window. These icons become active once more than one session is open in the work area.

Figure 2.8. Windows Menu

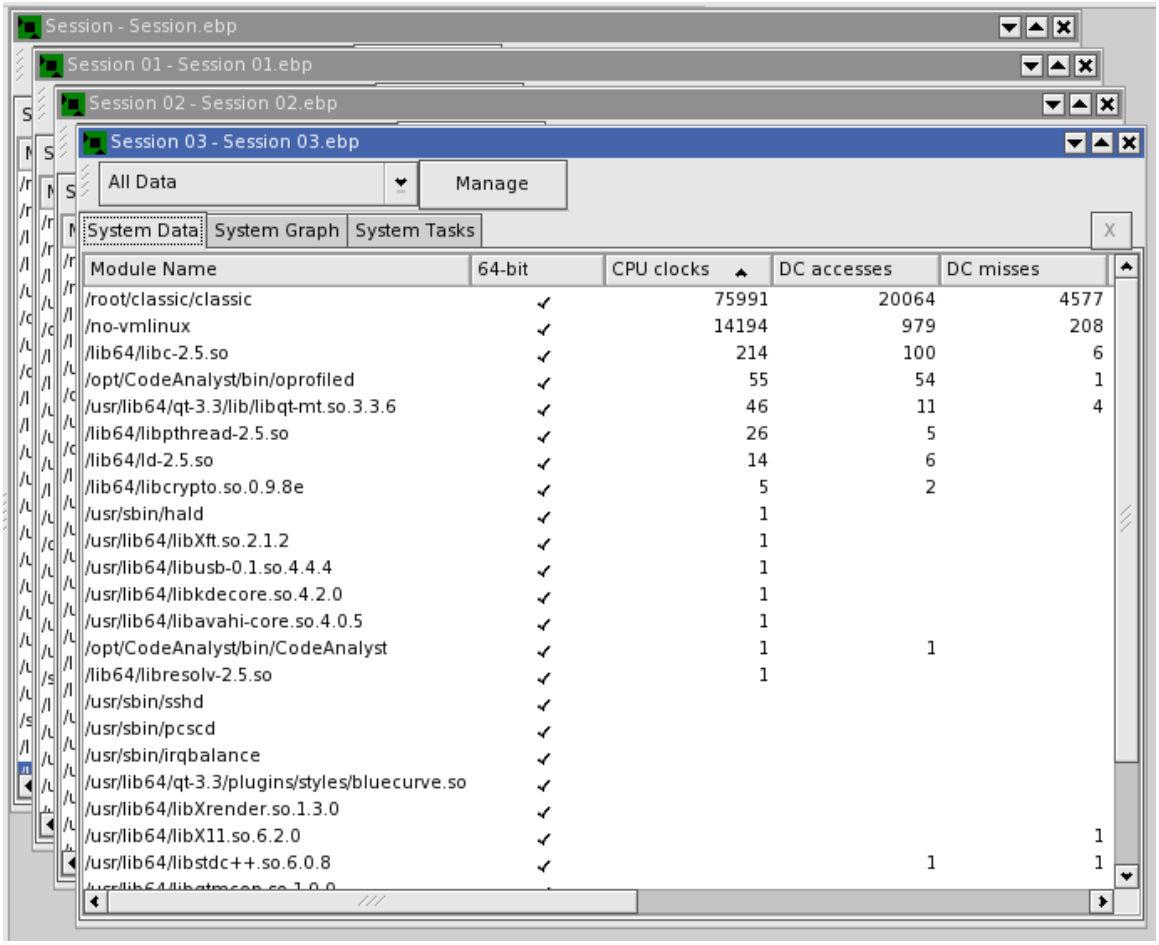


Menu Command	Description
Cascade	Displays open windows as overlapping and cascading downward from the upper left area of the work area.
Tile	Displays the open windows in a non-overlapping, tiled fashion.
Close All	Closes all open windows.
Session [ ]	Displays open windows. A check mark indicates the current window with focus. Each session is assigned a number and extension to differentiate between sessions with the same name File extensions further define the file as timer-based (.tbp), event-based (.ebp), or as a session imported from OProfile (import.tbp)

2.2.9.1. Cascading Session Panes

When two or more sessions are open, session panes can be cascaded. Following are examples of cascading panes.

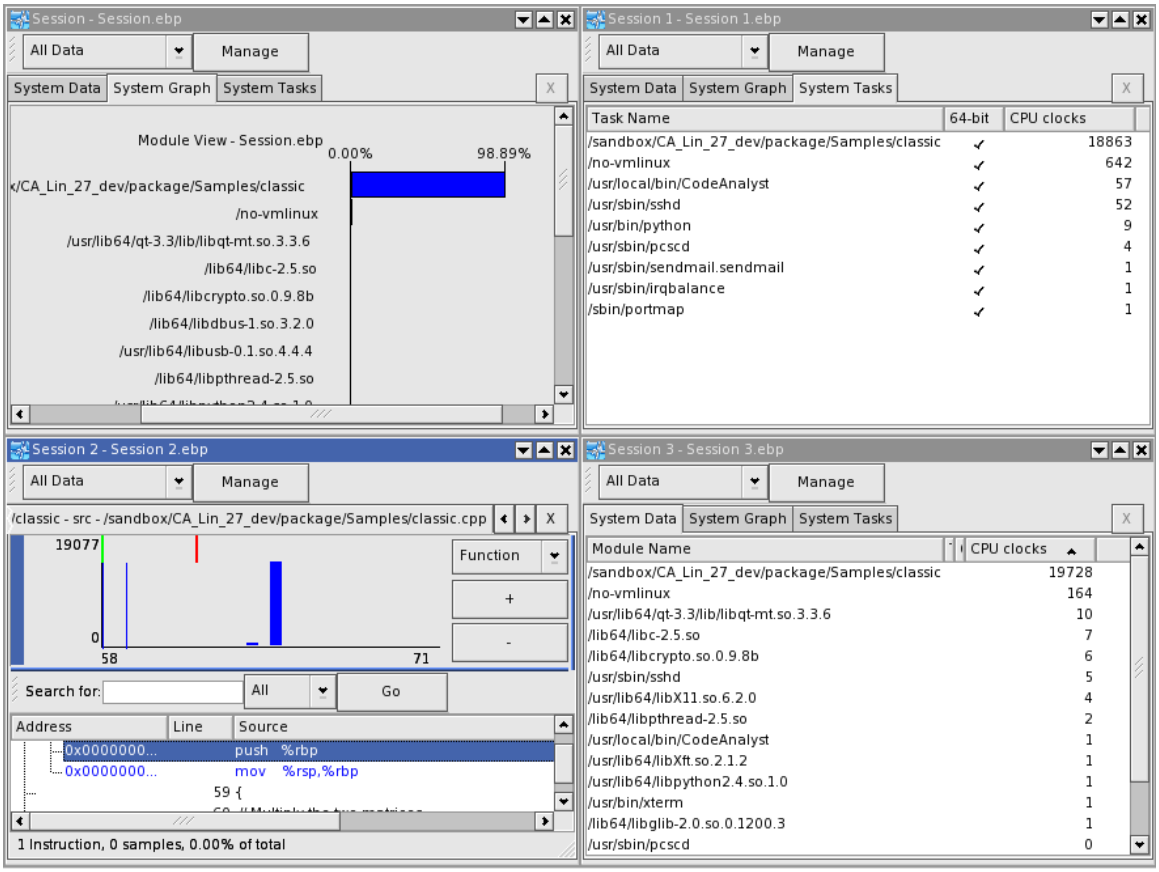
Figure 2.9. Cascading session panes



2.2.9.2. Tiling Session Panes

When two or more sessions are open, session panes can be tiled for viewing more than one pane at a time. Following are examples of tiled panes.

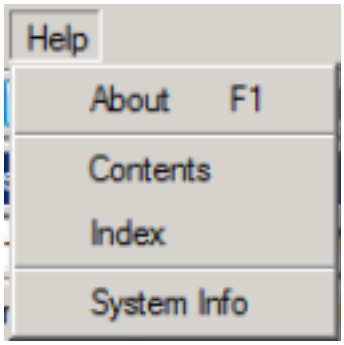
Figure 2.10. Tiling session panes



2.2.10. Help Menu

The Help menu displays the following.

Figure 2.11. Help menu



The commands available in the Help menu are:

Menu Command	Description
About F1	Displays the application flash screen and the version information.
Help	Displays the CodeAnalyst User's Manual.
System Info	Displays the System Information dialog box and reports system characteristics such as processor model, operating system version, memory characteristics, and video resolution.

2.2.11. Data and Source Display

Beginning with CodeAnalyst 2.6, new user interface elements are available, located above the tabbed panels. Results are shown in the form of data tables, graphs, and annotated source code. The main window opens with no data. Once a profile (data collection) configuration has been selected from the drop-down list and data has been collected, results are displayed. The time-based and event-based sampling session window each contain three initial tabbed panels:

- System Data
- System Graph
- System Tasks

Additional interface elements are available. A drop-down list of preset views allows quick selection of a new view. A view is a set of event data and computed performance measurements that display together in a table or a graph. The All Data view is the default view, which shows all available data in columns. Each column represents a distinct type of data or a computed performance measurement. Use the Section 7.3, “View Management” dialog box to change view configurations.

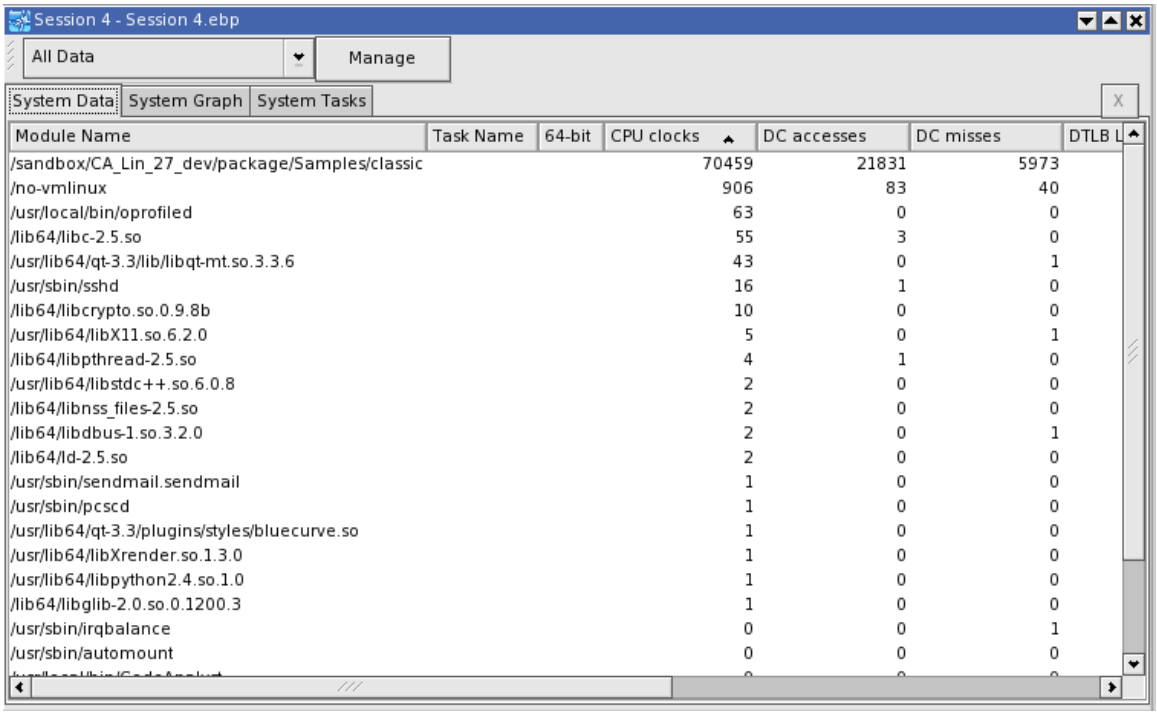
Source code is organized to allow for drilling down at the source level using an expandable tree of functions. The source for the selected function is displayed while all other functions are hidden. The source functions are viewed using the + sign and are hidden using the - sign.

2.2.11.1. System Data Tab

The System Data tab lists the modules and sample counts in descending sample count order. For multiprocessor systems, samples are shown for each processor. The 64-bit column distinguishes if samples are 32-bit modules or 64-bit modules. The System Graph tab displays a graphical view of the system data samples.

Double-click on a module name to drill down to a view of the TBP or EBP samples in a specific module.

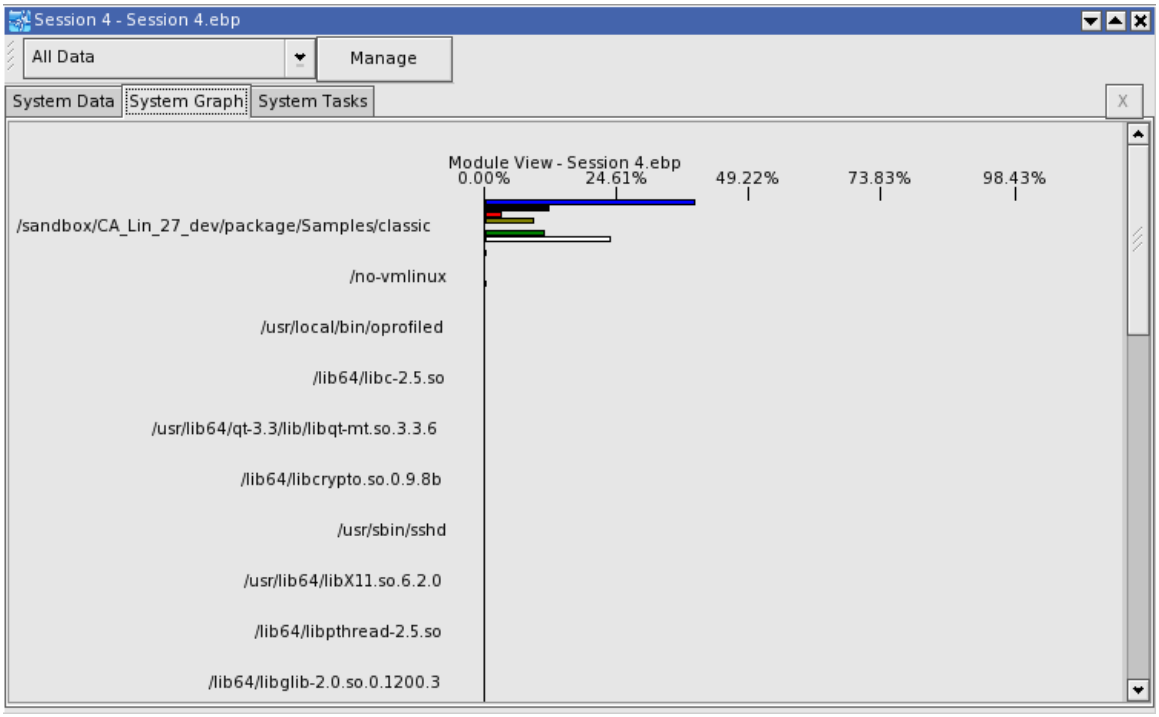
Figure 2.12. System Data tab



2.2.11.2. System Graph Tab

The System Graph tab provides a visual display of the sample counts per module in a horizontal bar graph in descending order of sample count. This view provides a quick view of the most processor intensive modules. Multiprocessor systems can show events for each processor (see following figure) or a cumulative bar for each event. Double-click on a module to drill down to the TBP or EBP samples within the selected module.

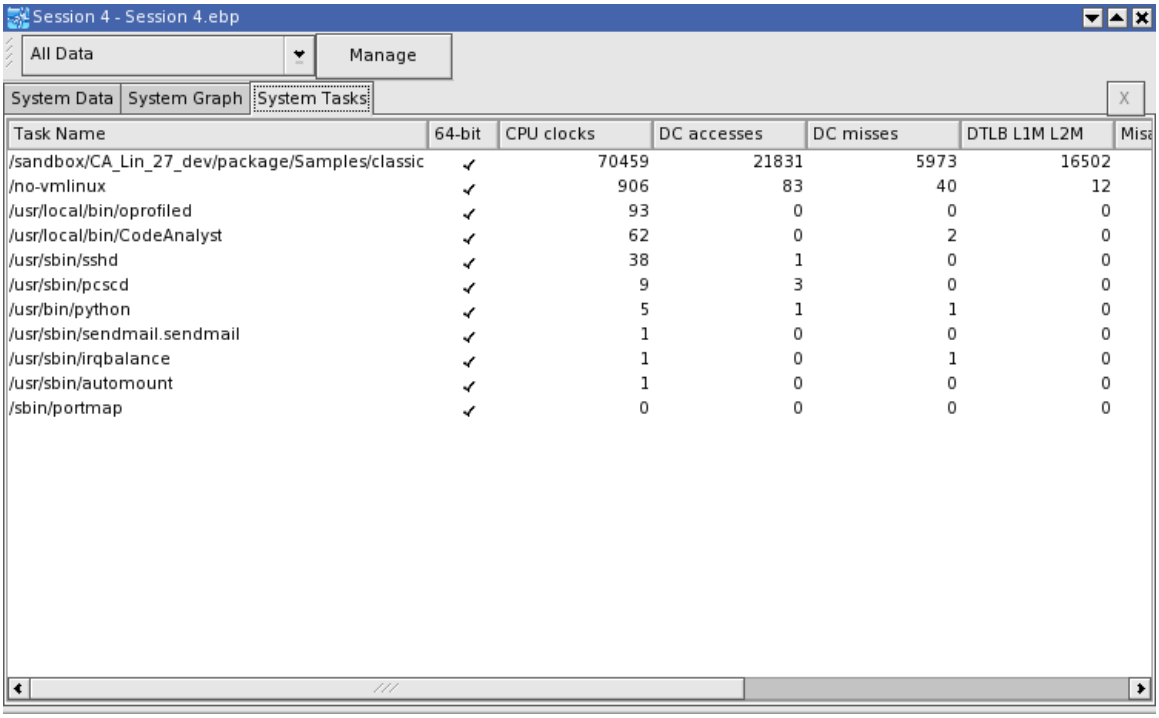
Figure 2.13. System Graph tab



2.2.11.3. System Tasks Tab

The System Tasks tab displays the task name, total samples and percentage, and single (or multi-core) events. Double-clicking on a module drills down to the TBP or EBP samples within the selected module. Each column has an up/down arrow for sorting the column in ascending or descending order.

Figure 2.14. System Tasks tab

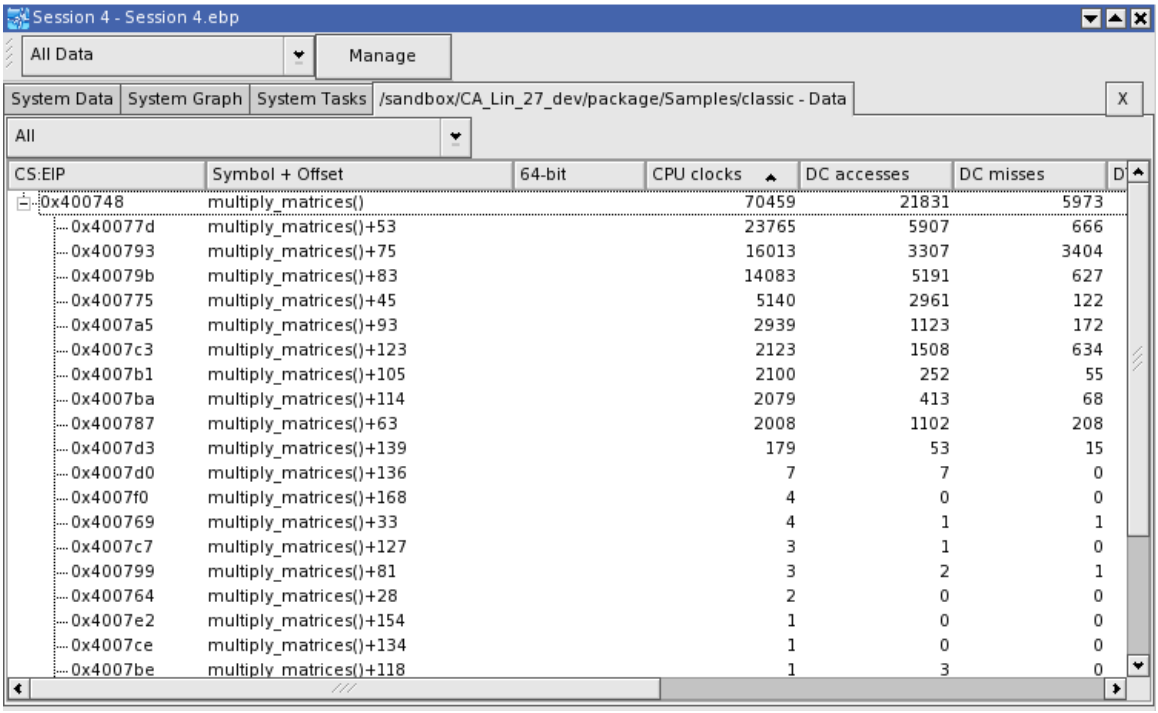


Task Name	64-bit	CPU clocks	DC accesses	DC misses	DTLB L1M L2M	Misa
/sandbox/CA_Lin_27_dev/package/Samples/classic	✓	70459	21831	5973	16502	
/no-vmlinux	✓	906	83	40	12	
/usr/local/bin/oprofiled	✓	93	0	0	0	
/usr/local/bin/CodeAnalyst	✓	62	0	2	0	
/usr/sbin/sshd	✓	38	1	0	0	
/usr/sbin/pcscd	✓	9	3	0	0	
/usr/bin/python	✓	5	1	1	0	
/usr/sbin/sendmail.sendmail	✓	1	0	0	0	
/usr/sbin/irqbalance	✓	1	0	1	0	
/usr/sbin/automount	✓	1	0	0	0	
/sbin/portmap	✓	0	0	0	0	

2.2.11.4. Single Module Data Tab

The Single Module Data tab drills down into a single module. It illustrates how the data samples are distributed within a module. Like the System Data tab, this view shows the distribution of samples per processor in a multiprocessor system and between 32-bit and 64-bit modules in a 64-bit system (64-bit column). The data samples can be expanded and collapsed around the available symbols. Right-clicking on an address will bring up a choice of viewing the information in a graph view. If debug information is available, double-clicking on an address navigates to the Source View. If debug information is not available, double-clicking on an address navigates to the Assembly View.

Figure 2.15. Single Module Data tab

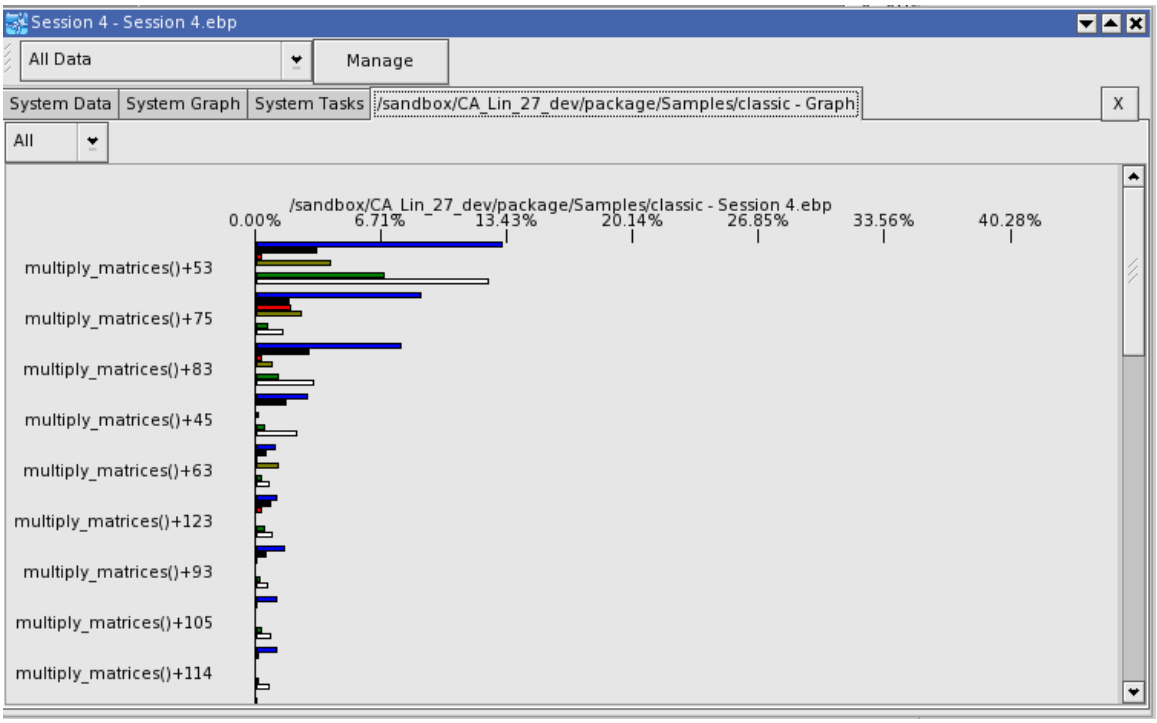


CS:EIP	Symbol + Offset	64-bit	CPU clocks	DC accesses	DC misses
0x400748	multiply_matrices()		70459	21831	5973
0x40077d	multiply_matrices()+53		23765	5907	666
0x400793	multiply_matrices()+75		16013	3307	3404
0x40079b	multiply_matrices()+83		14083	5191	627
0x400775	multiply_matrices()+45		5140	2961	122
0x4007a5	multiply_matrices()+93		2939	1123	172
0x4007c3	multiply_matrices()+123		2123	1508	634
0x4007b1	multiply_matrices()+105		2100	252	55
0x4007ba	multiply_matrices()+114		2079	413	68
0x400787	multiply_matrices()+63		2008	1102	208
0x4007d3	multiply_matrices()+139		179	53	15
0x4007d0	multiply_matrices()+136		7	7	0
0x4007f0	multiply_matrices()+168		4	0	0
0x400769	multiply_matrices()+33		4	1	1
0x4007c7	multiply_matrices()+127		3	1	0
0x400799	multiply_matrices()+81		3	2	1
0x400764	multiply_matrices()+28		2	0	0
0x4007e2	multiply_matrices()+154		1	0	0
0x4007ce	multiply_matrices()+134		1	0	0
0x4007be	multiply_matrices()+118		1	3	0

2.2.11.5. Single Module Graph Tab

The Single Module Graph tab provides a view of the sample distribution within a single module. Right-click for a choice to navigate back to the Single Module Data View. If debug information is available, double-clicking will navigate to the Source View. Otherwise, double-clicking will navigate to the Assembly View.

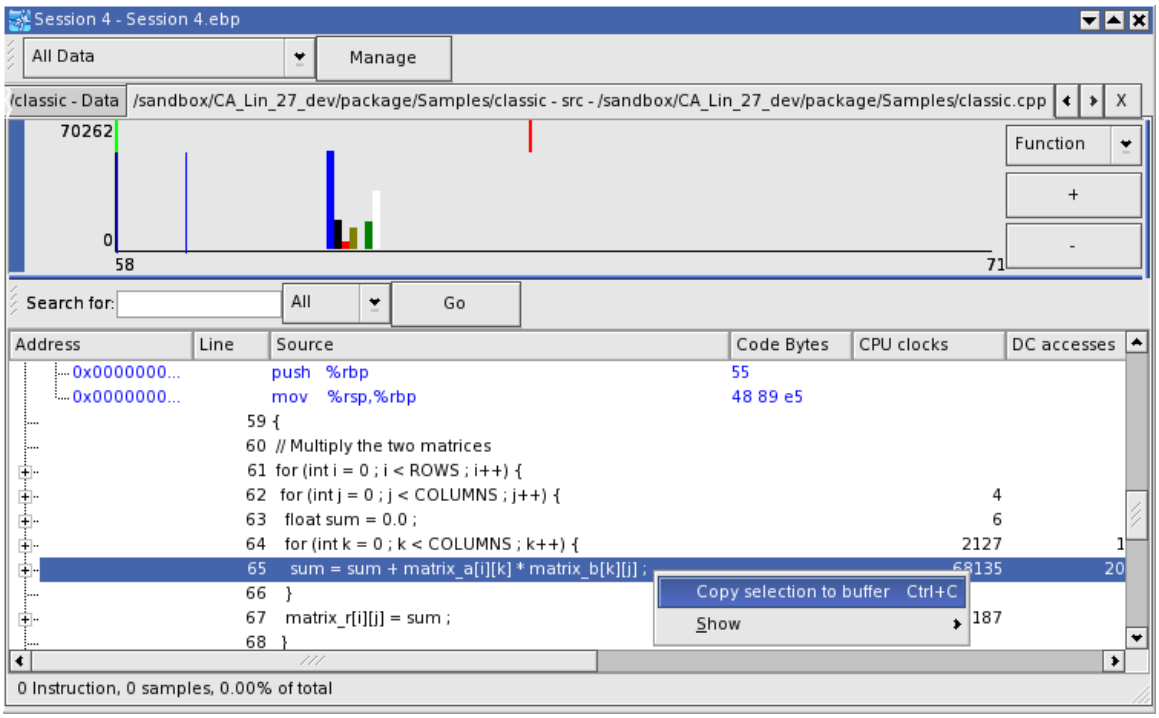
Figure 2.16. Single Module Graph tab



2.2.11.6. Profiling Session Source Tab

The Profiling Session Source tab displays the source with the sample count for each source line. The source line can be expanded or collapsed to show or hide the assembly instructions that are associated with the source line. Right-click to select **copy the selection to a buffer** (the clipboard). The information may then be pasted into another document.

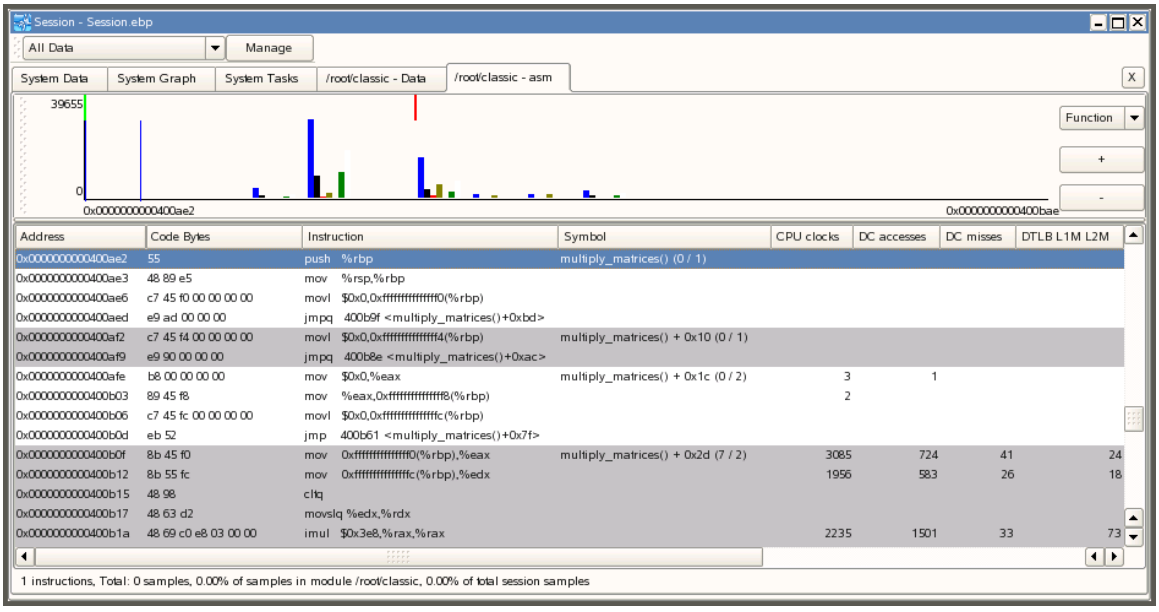
Figure 2.17. Profiling Session Source tab



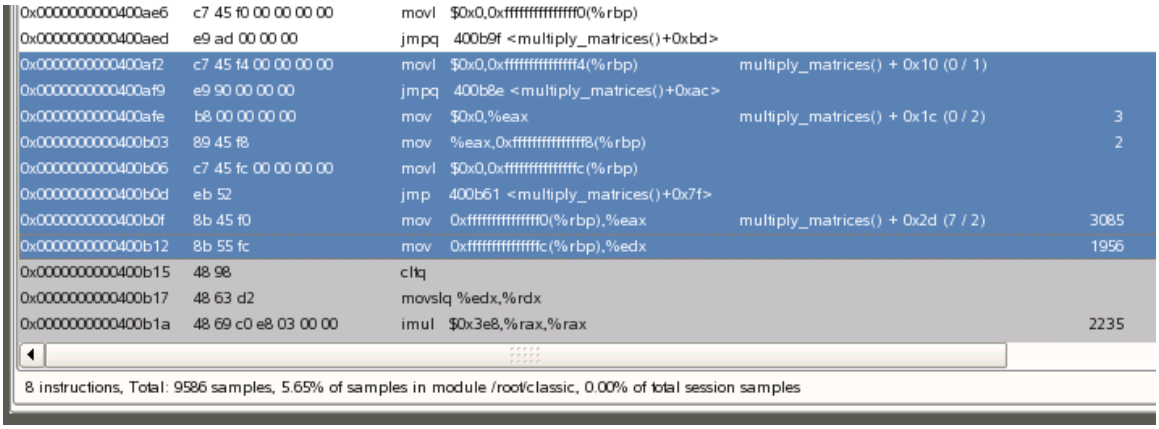
2.2.11.7. Assembly Tab

The Assembly tab displays the assembly instructions around an address selected from the Single Module View.

Figure 2.18. Assembly tab

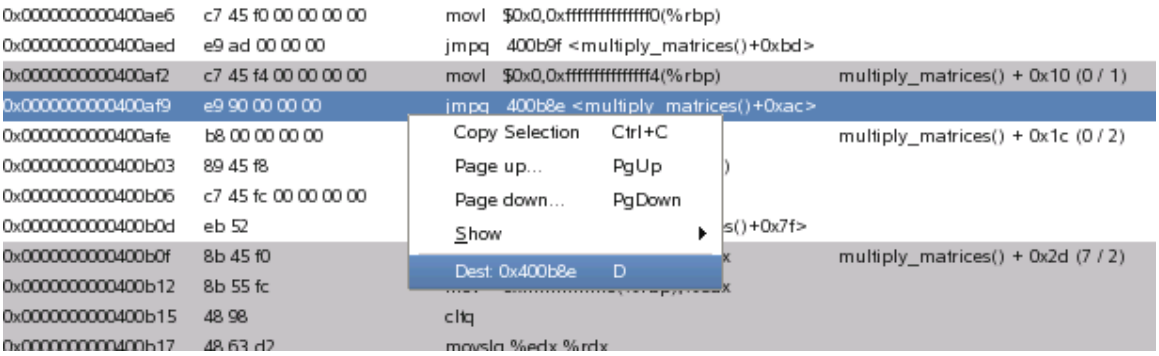


When selecting multiple instructions on the Assembly tab, AMD CodeAnalyst displays a summary of the selection in the status bar of the Assembly window.



The sample summary is also available for the source level.

Press the **Page Up** and **Page Down** to view 200 bytes above or below the highlighted instruction.



In Disassembly View, each basic block is shown by interleaving different background colors of white and gray. Users can navigate through code execution paths from one basic block to the previous or the next basic block. Right-clicking at the beginning of a basic block, opens a menu that lists the source addresses that are usually the destination address of a control transfer instruction in some basic blocks. Right-click at the end of a basic block to open a list with the destination address of the control transfer instruction (see previous figure).

2.2.12. Code Density Chart

Selecting "Show code density chart" under Tools > CodeAnalyst Options > Toolbars displays a code density chart on the Assembly (asm) tab and on the Source View tab. The chart on the asm tab shows the number of samples relative to the location within the module, function, a user-specified area, or the currently shown assembly. The chart for the Source View tab shows the number of samples relative to the source lines in the whole source file, function, a defined region, or the currently shown source. The initial zoom level is at the function level and can also be shown at module, partial, and current view.

Figure 2.19. Code Density Chart

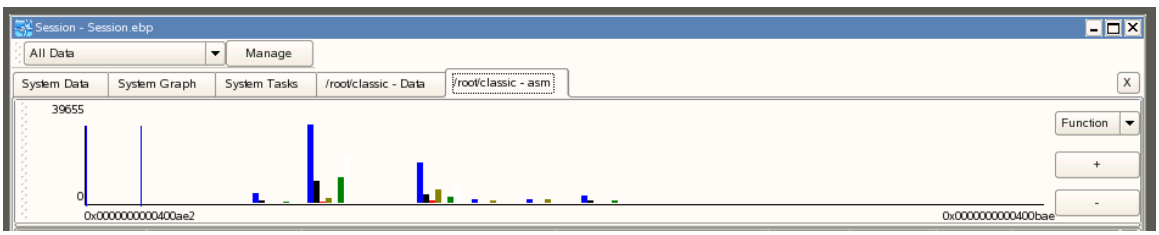


Figure 2.20. A drop-down list provides choices for selecting code density.

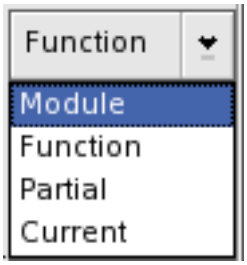
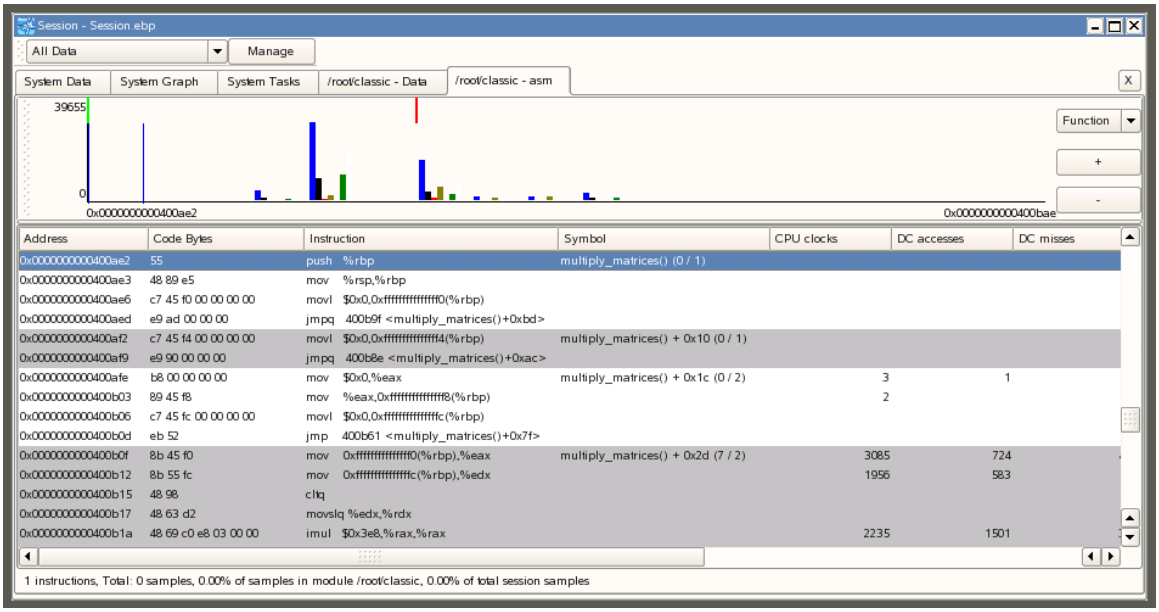


Figure 2.21. An assembly view with Current samples displayed.



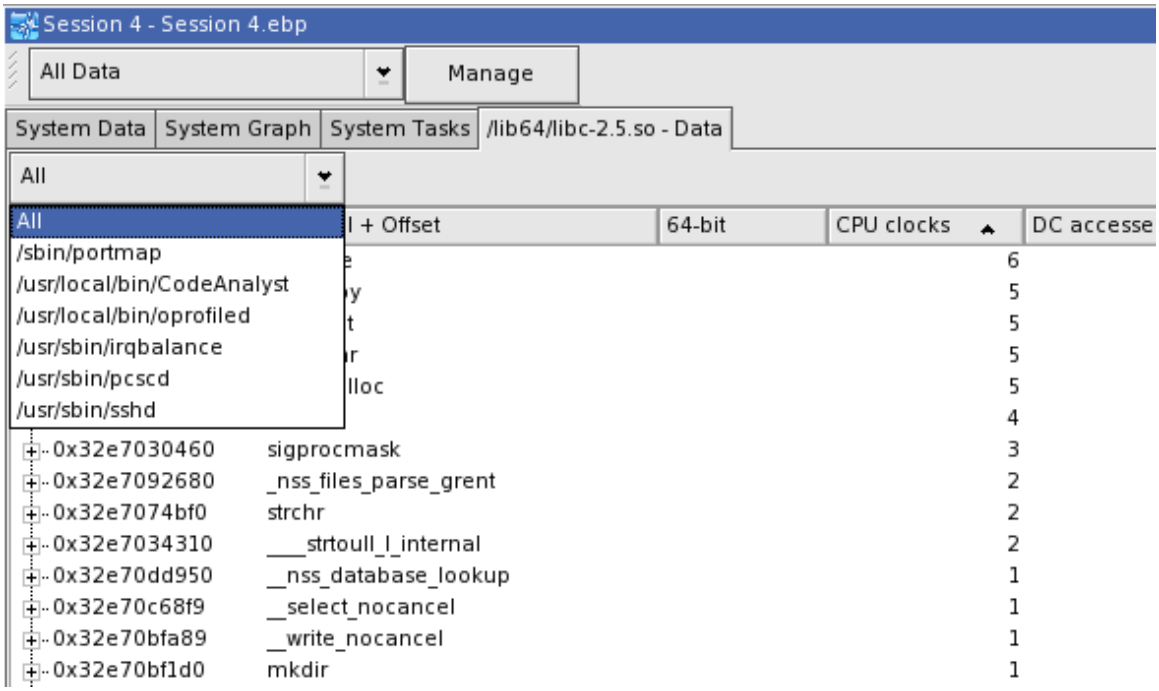
2.2.13. System Data Interface Elements

The GUI provides interface elements on certain session tabs to control the display and use of **Task** and the aggregation/separation of data by task.

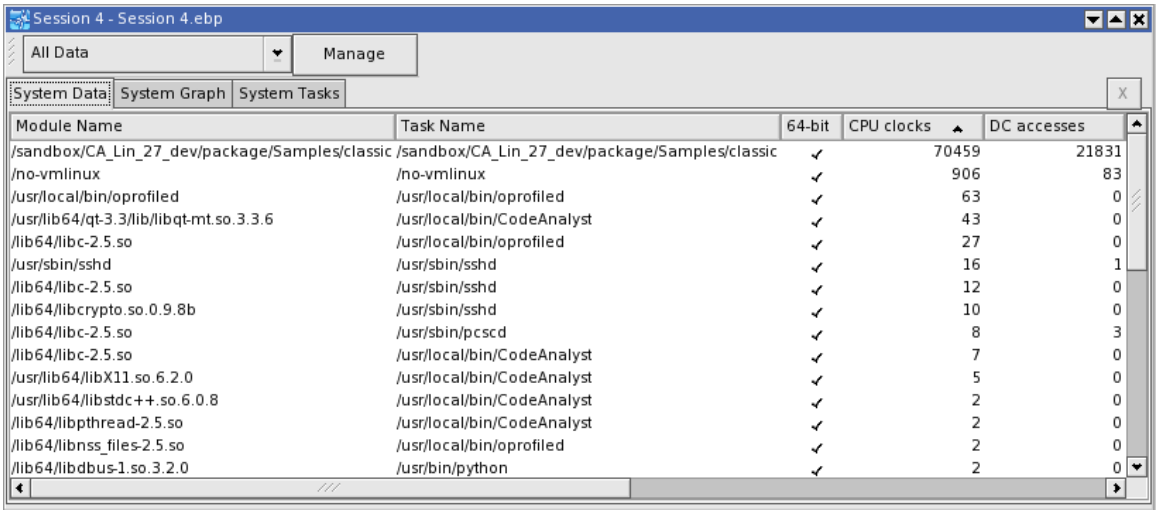
2.2.14. Displaying Tasks

Task can be shown in the System Data view. Use the check box **Separate Task** in **View Management** to display or hide task name. A drop-down list also appears to allow faster access to specific task. Tasks may be selected separately or all tasks may be shown by selecting **All**.

Figure 2.22. Displaying Tasks



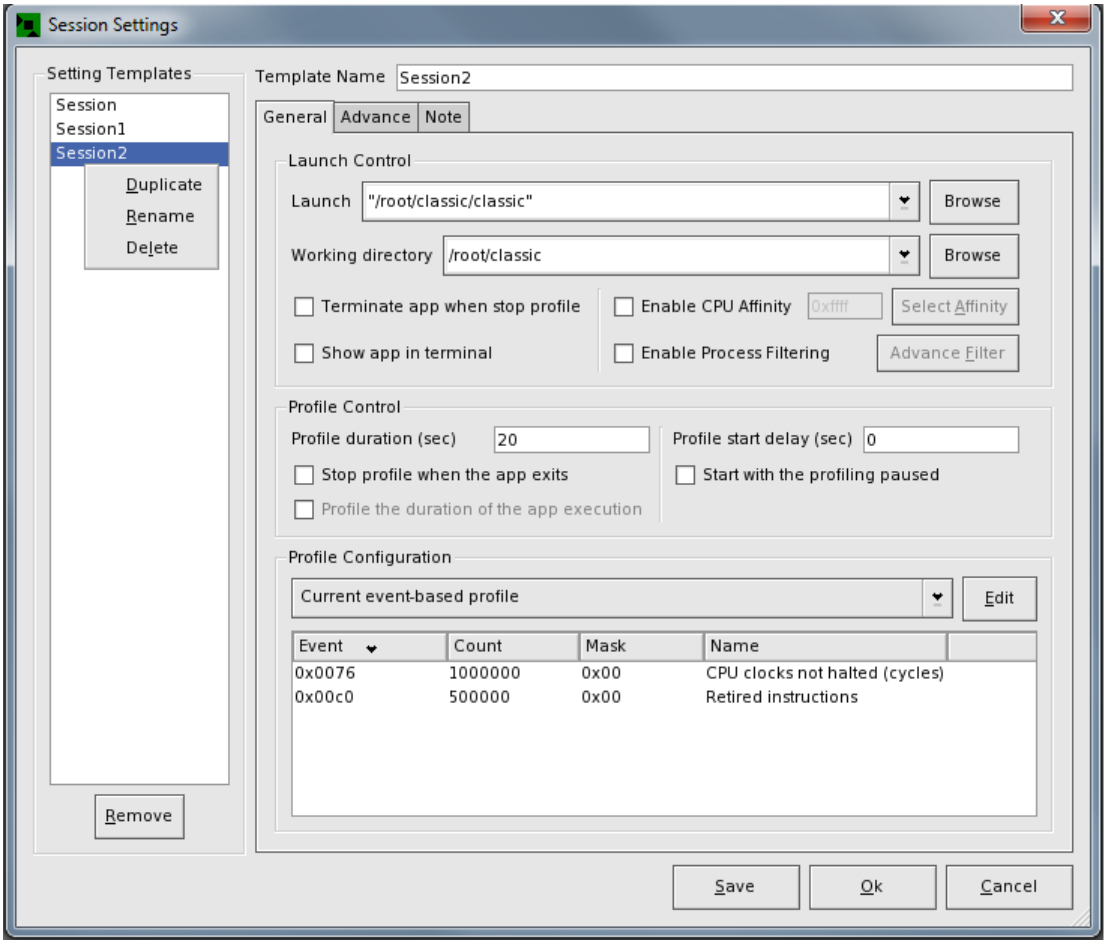
Display task name



2.2.15. Session Settings

The Session Settings dialog box sets and supports changes to the session parameters. Please see Section 2.7, “Session Settings” for more detail.

Figure 2.23. Session Settings



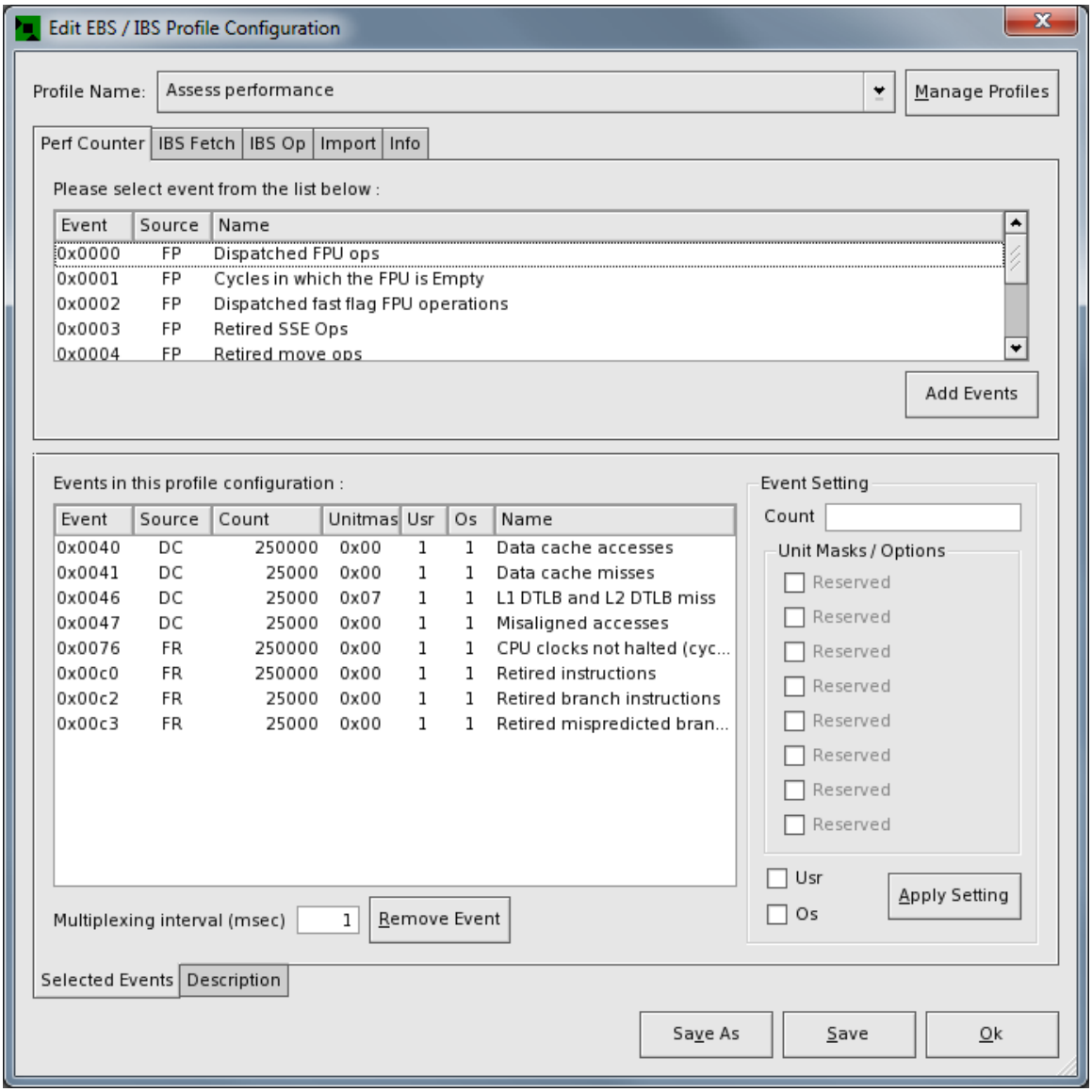
2.2.16. Edit Event Configuration

The **Edit** button is located in two dialog boxes:

- Session Settings
- Configuration Management

Clicking Edit opens the "Edit Event Configuration" dialog box. For details on using the dialog box, read Section 4.3, “Edit Event-based and Instruction-based Sampling Configuration”.

Figure 2.24. Edit event configuration



2.2.17. View Management Dialog Box

The Section 7.3, “View Management” dialog box allows customization of views. After performance data is collected (or imported), it is managed as a pool of available performance data. A CodeAnalyst view specifies the kinds of data to be displayed. This feature allows users to choose and focus on performance information that is the most relevant to the issue under investigation. For further explanation, refer to Chapter 7, *View Configuration*

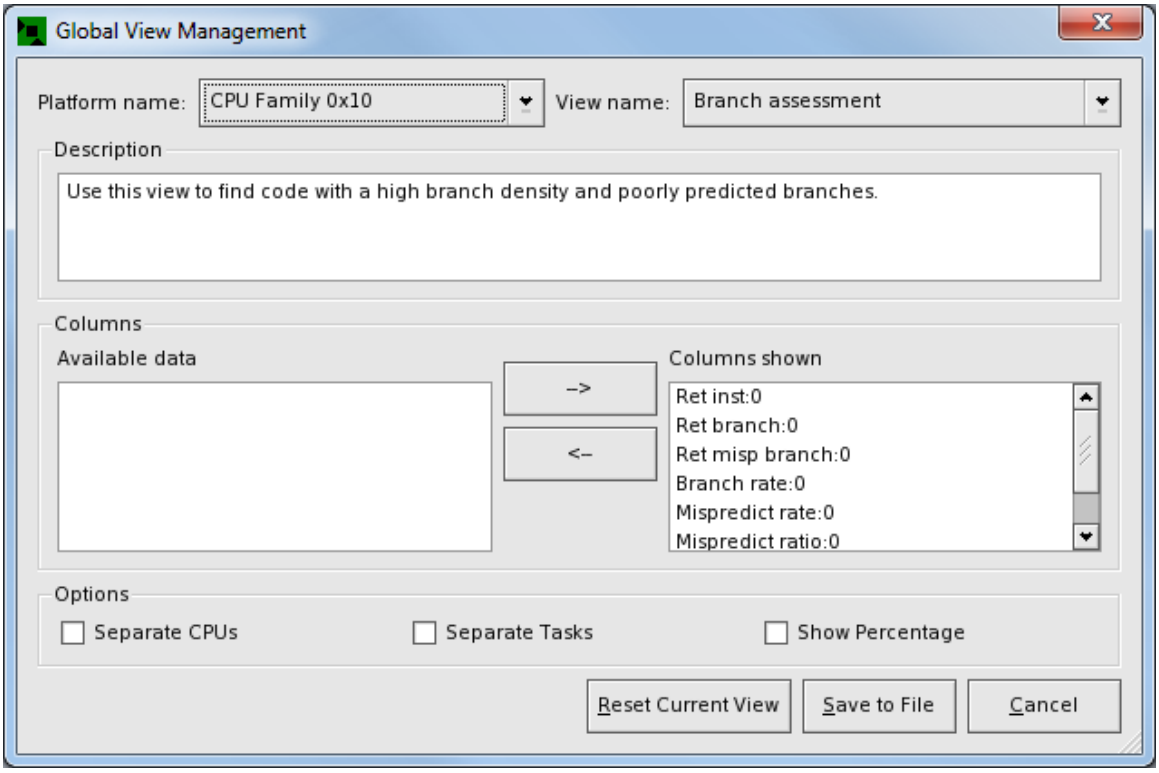
There are two types of View Management dialog: methods:

- **Local View Management Dialog** is opened by clicking the **Manage** button 

displayed in an open session window.
- **Global View Management Dialog** is opened by selecting the **Views** icon  from the toolbar or from the **Tools** menu.

For details using this dialog box, go to Section 7.3, “View Management”

Figure 2.25. Global View Management



2.2.18. Configuration Management Dialog Box

Performance data collection is controlled by profile configurations. A profile configuration specifies basic run control parameters, types of data to collect, and how data is to be collected. Certain configurations can be "managed" by the user to create new profile configurations. Specifications for basic run control, types of data collected, and how data is to be collected are determined through the configuration's profile. Predefined profile configurations and user-defined profile configurations are found in the **Configuration Management** dialog box or in the toolbar list of profile configurations.

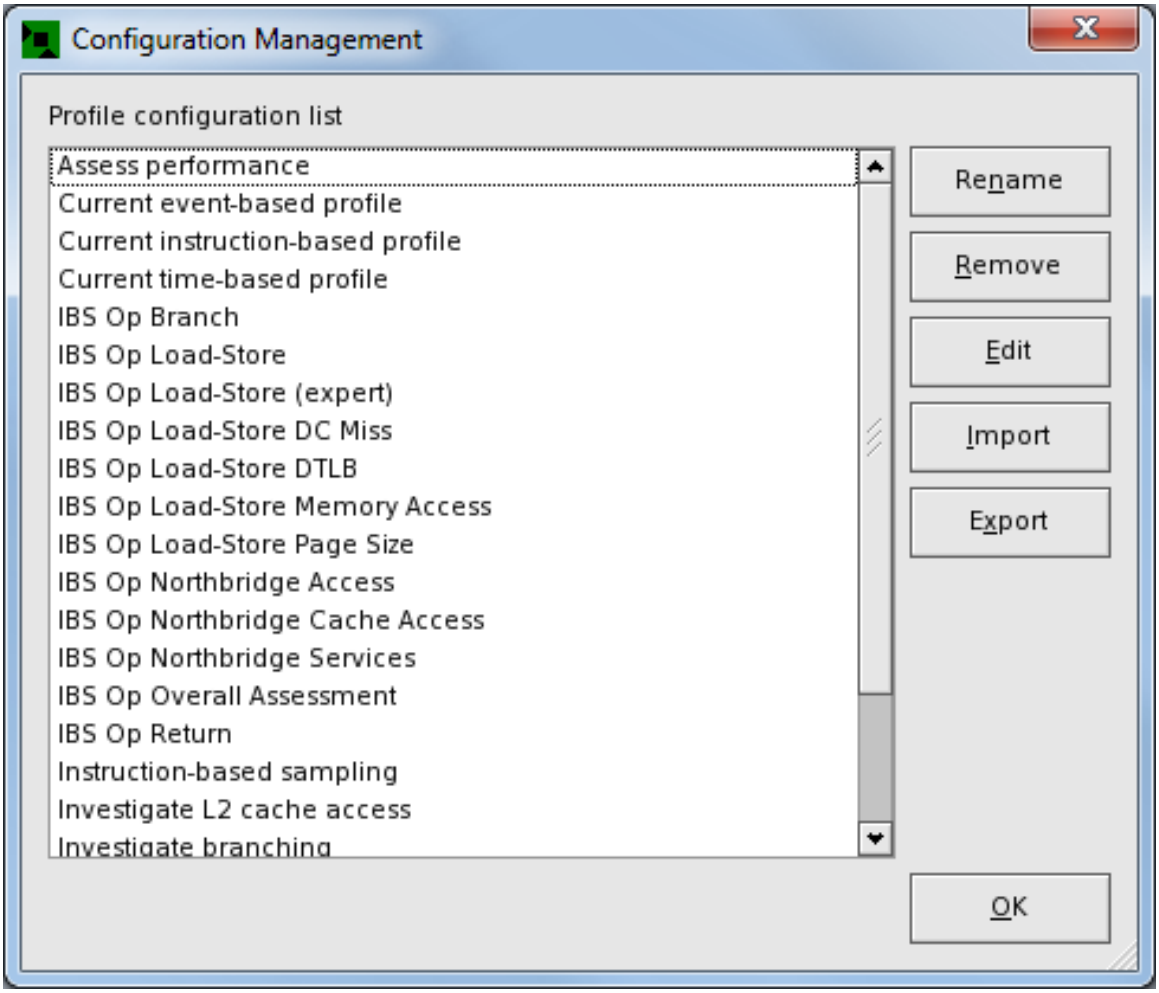
Configuration management allows for customizing existing profiles and for creating new ones. Configurations for both profiles and views are stored in files when CodeAnalyst is not running. Saving in this manner allows for easy sharing of files. Each user-created configuration is permanently stored as an file until it is removed by using the **Remove** button. See Section 4.5, “Manage Profile Configurations” for additional details.

Use the **Configuration** icon  to open the dialog box.

2.2.18.1. Current-Type Profiles

The profile configuration list contains three "Current" configurations (Current time-based profile, Current event-based profile, Current instruction-based profile). These configurations are considered as appropriate starting points for customization. See Section 4.5, “Manage Profile Configurations” for detailed information.

Figure 2.26. Configuration management



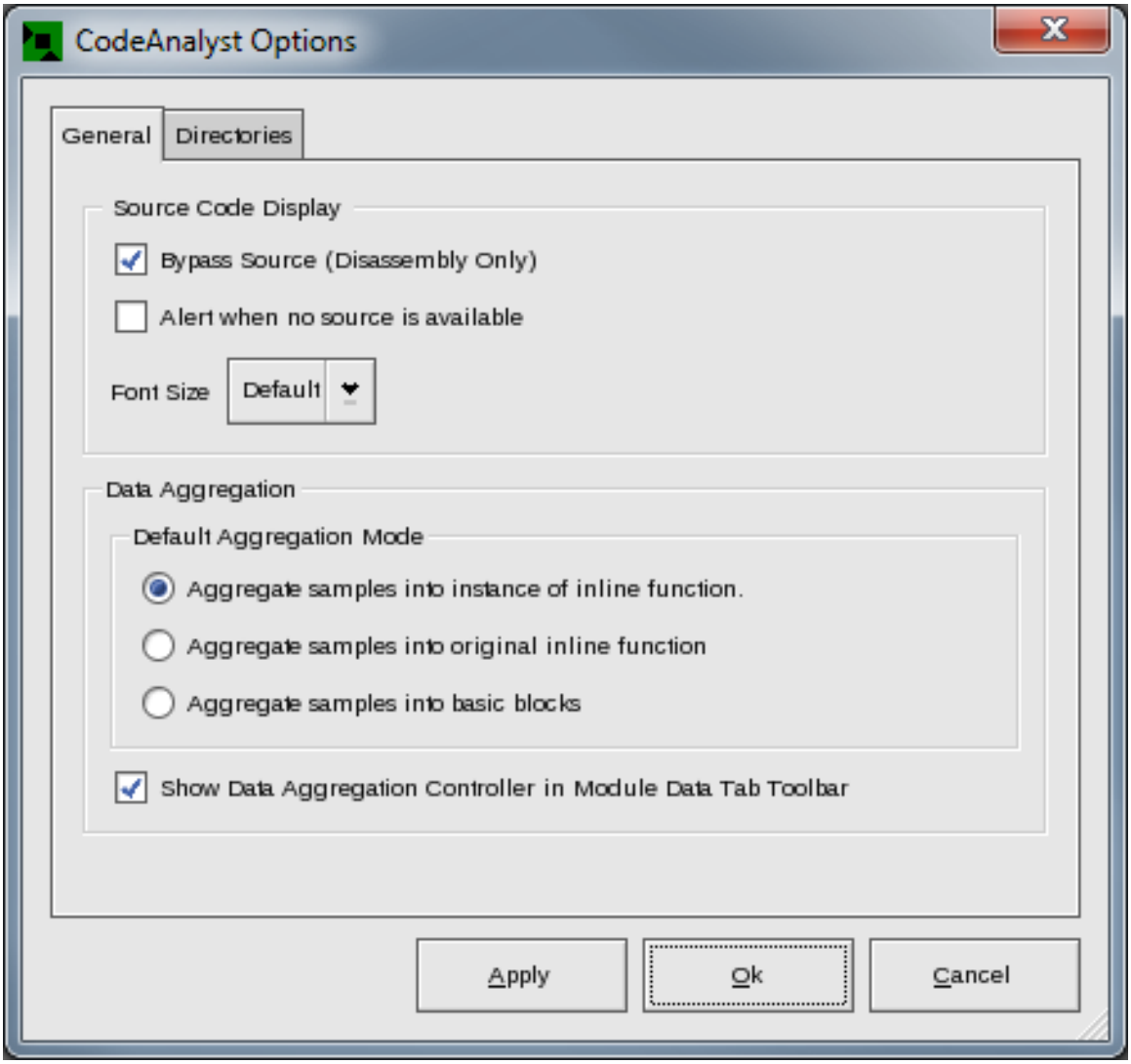
2.2.19. CodeAnalyst Options Dialog Box

The CodeAnalyst options dialog box has tabs for setting the following options.

- General—Use to assign types of source code displayed, module enumeration, and hot keys.
- Directories—Select various director
- Oprofile—Select Oprofile plugin and Oprofile parameters

For details on using this dialog box, to to Section 2.3, “CodeAnalyst Options”.

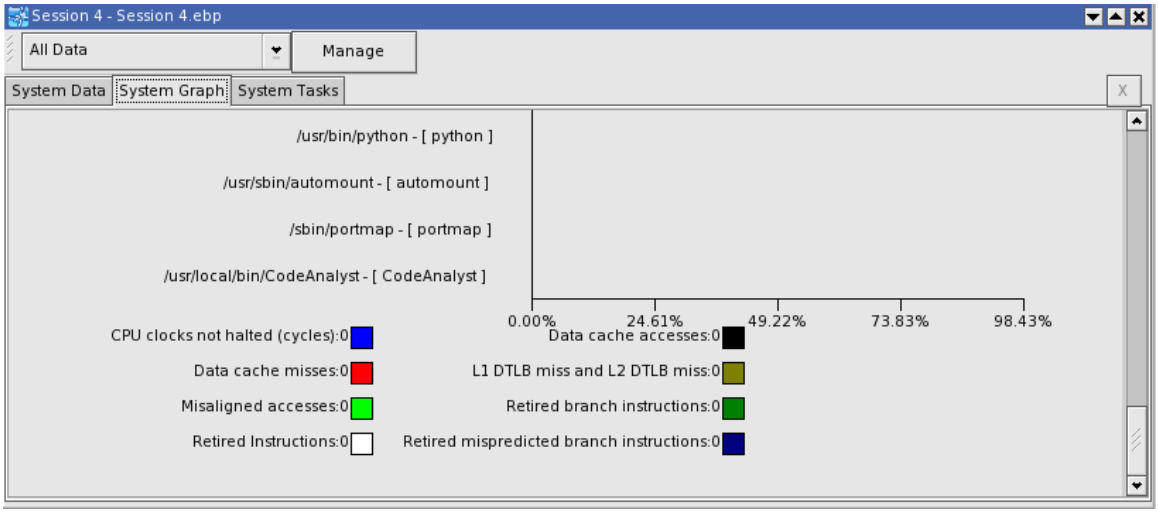
Figure 2.27. CodeAnalyst Options



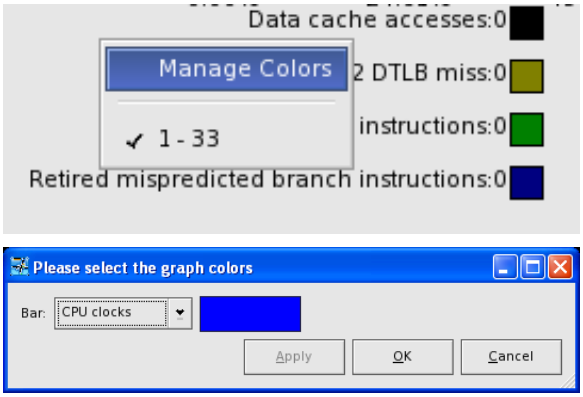
2.2.20. Manage Colors

Specific colors are assigned to the bars in graphs to designated processors and events. The color key is located at the bottom of the System Graph tab. The following example shows the color key, event-based profile. Right-click in this area to open the list of samples.

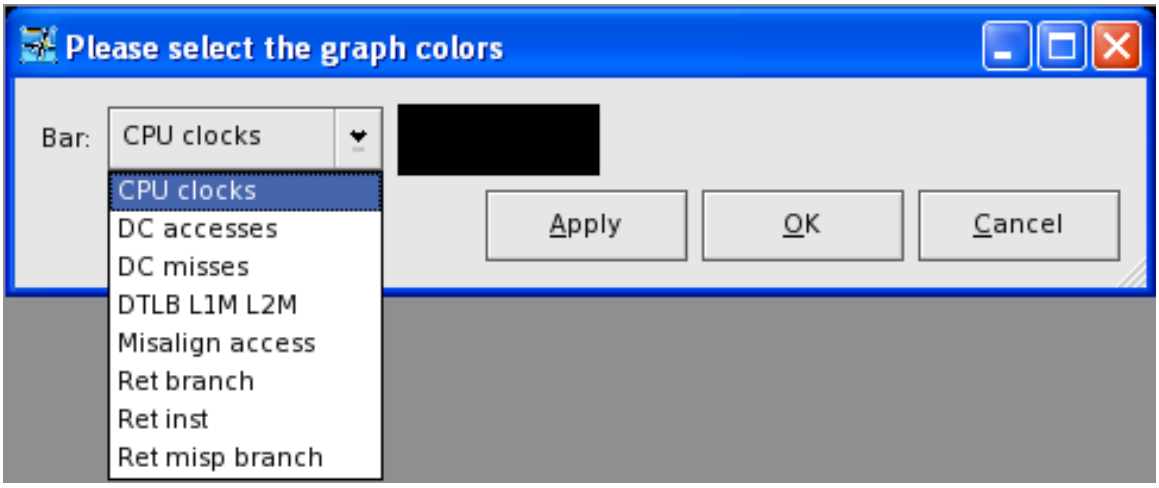
Figure 2.28. Managing System Graph colors



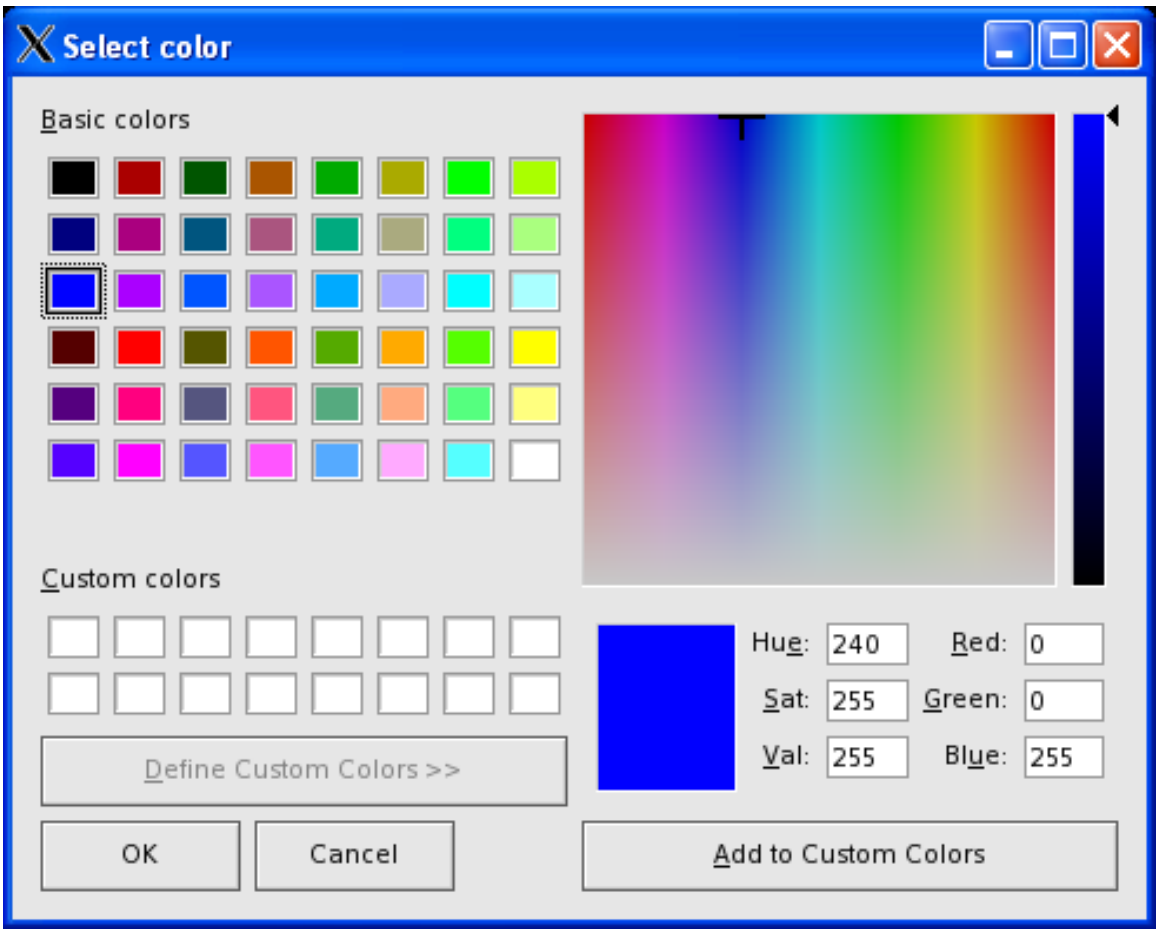
Right-click on a box in the color key to obtain a pop-up menu. Select Manage Colors from the menu in order to change the color. A color selection drop-down list displays



The drop-down list contains the names of the events or processors in the graph. Choose a name from the list and click the color block to change the color for a particular graph element. Following is an example list.



Double-click on the color bar to the right of the list to open the color palette.



2.2.21. Profiling Java Applications

Profiling of Java applications is supported. A session is created to hold the performance data. Users can view the samples at the function, source, or assembly level. IBM and Sun versions of the Java Virtual Machine (JVM) are supported.

To profile Java applications, choose **Tools > Project Options**. The Project Options dialog box displays.

To launch the Java application from within AMD CodeAnalyst:

1. Type the path in the Working Directory and Launch App fields on the General tab.
2. The Java command line has to include **-agentpath:/usr/local/bin/libCAJVMTIA.so**. This also applies to any Java applications that are run outside AMD CodeAnalyst.

For example,

"/usr/bin/java" -agentpath:/usr/local/bin/libCAJVMTIA.so example1

where example1.class is in your working directory.

2.3. CodeAnalyst Options

CodeAnalyst options controls how AMD CodeAnalyst displays profile data and its toolbar and how it finds source and debug information. These options are persistent and are effective across projects and sessions. They affect the CodeAnalyst application as a whole.

CodeAnalyst options are changed using the CodeAnalyst Options dialog box. The dialog box contains following tabs:

- Section 2.3.1, “ General Tab”
- Section 2.3.2, “Directories Tab”

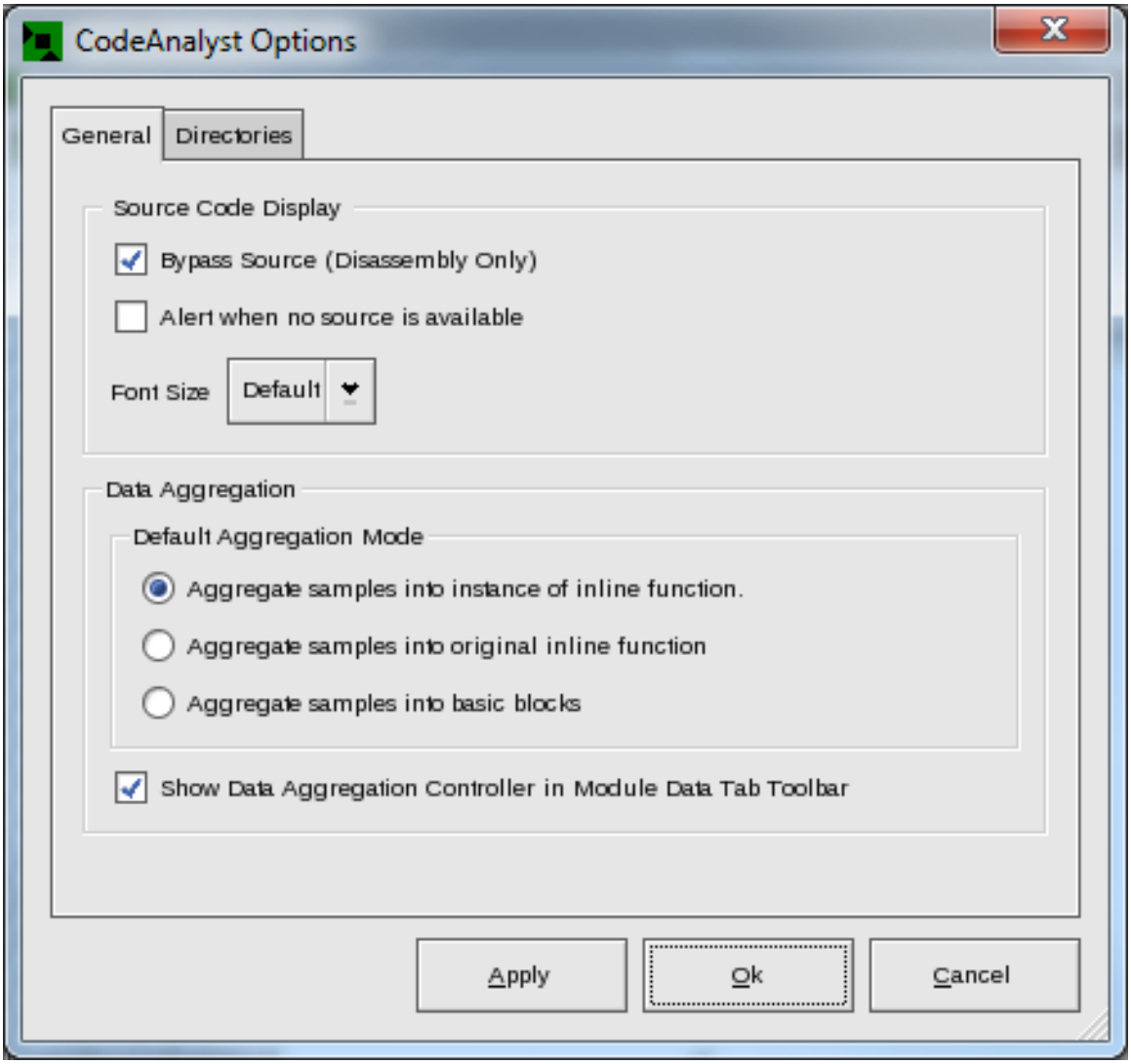
Open the CodeAnalyst Options dialog box by clicking the CodeAnalyst Options icon  in the toolbar. Or you may open the CodeAnalyst Options dialog box by selecting **Tools > CodeAnalyst Options** from the pull-down menu.

2.3.1. General Tab

The General tab controls the display of source code and disassembler instructions.

CodeAnalyst follows the usual Windows conventions for accepting or canceling changes to options. Clicking the **Apply** button activates the new options. Clicking the **OK** button activates the new options and closes the dialog box. Clicking the **Cancel** button closes the dialog box without applying changes.

Figure 2.29. CodeAnalyst Options



2.3.1.1. Source Code Display

If the **Bypass Source (Disassembly Only)** check box is selected, double-clicking a sample address in a module opens the Assembly View. Selecting the **Alert when no source is available** check box displays an alert message when CodeAnalyst cannot find the source for a module.

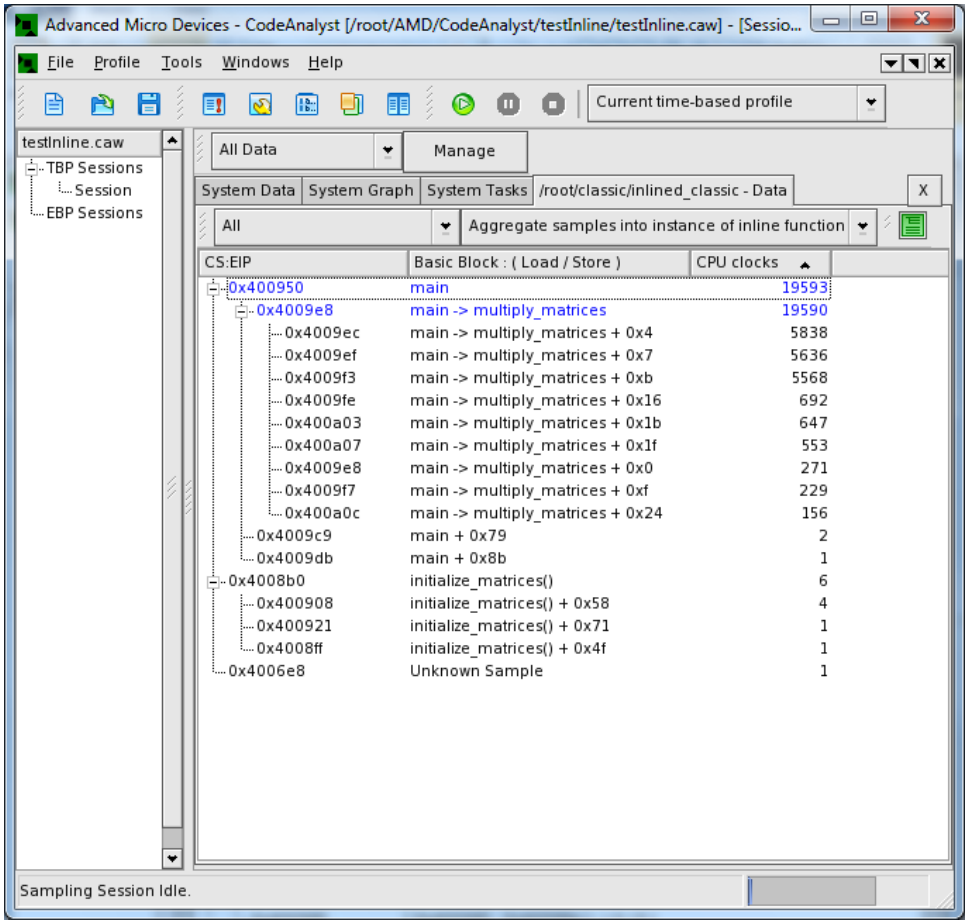
2.3.1.2. Data Aggregation

CodeAnalyst allows three different methods for aggregate data. Two modes are specifically designed to help analysis of in-line functions:

- Aggregate samples into instance of in-line function

  This is the default aggregation mode. When samples belong to an in-line instance, CodeAnalyst aggregates them into the caller function and uses blue text to identify the in-line instance together with in-line function name.

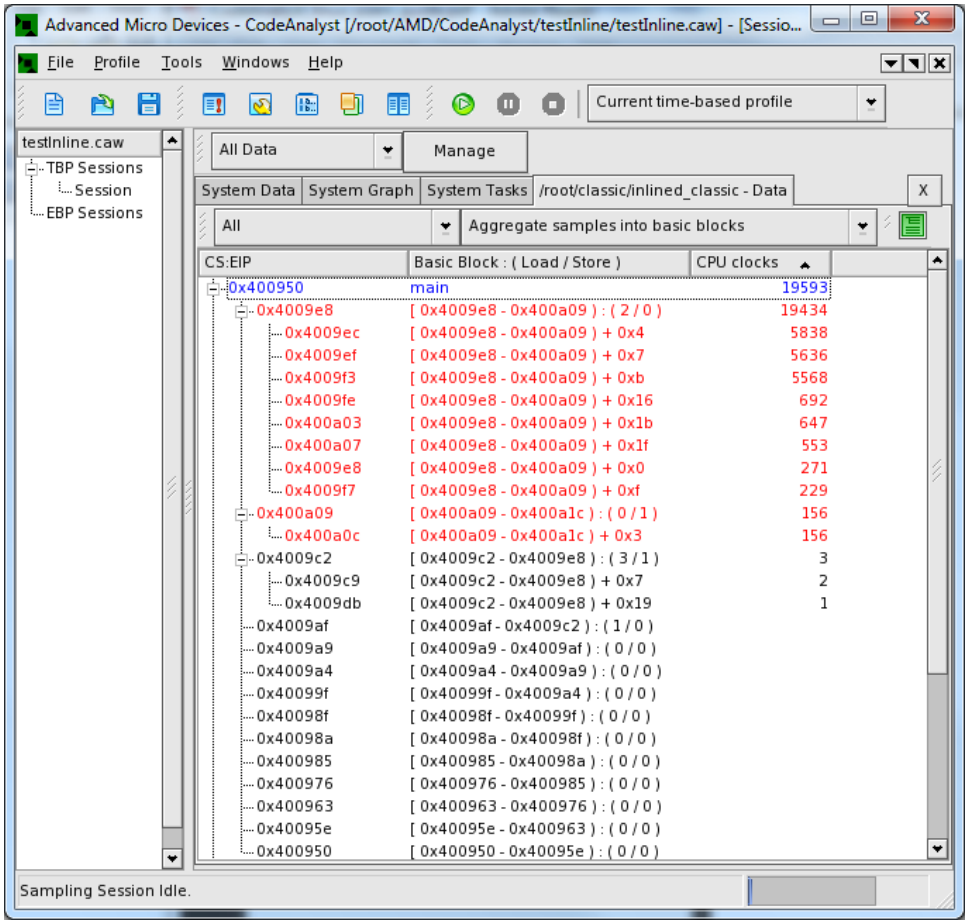
Figure 2.30. Aggregate samples into instance of in-line function



- Aggregate samples into original in-line function

  When samples belong to an in-line instance, CodeAnalyst aggregates them into each in-line instance. CodeAnalyst groups all in-line instances and lists them together under the in-line function, which is presented in red text.

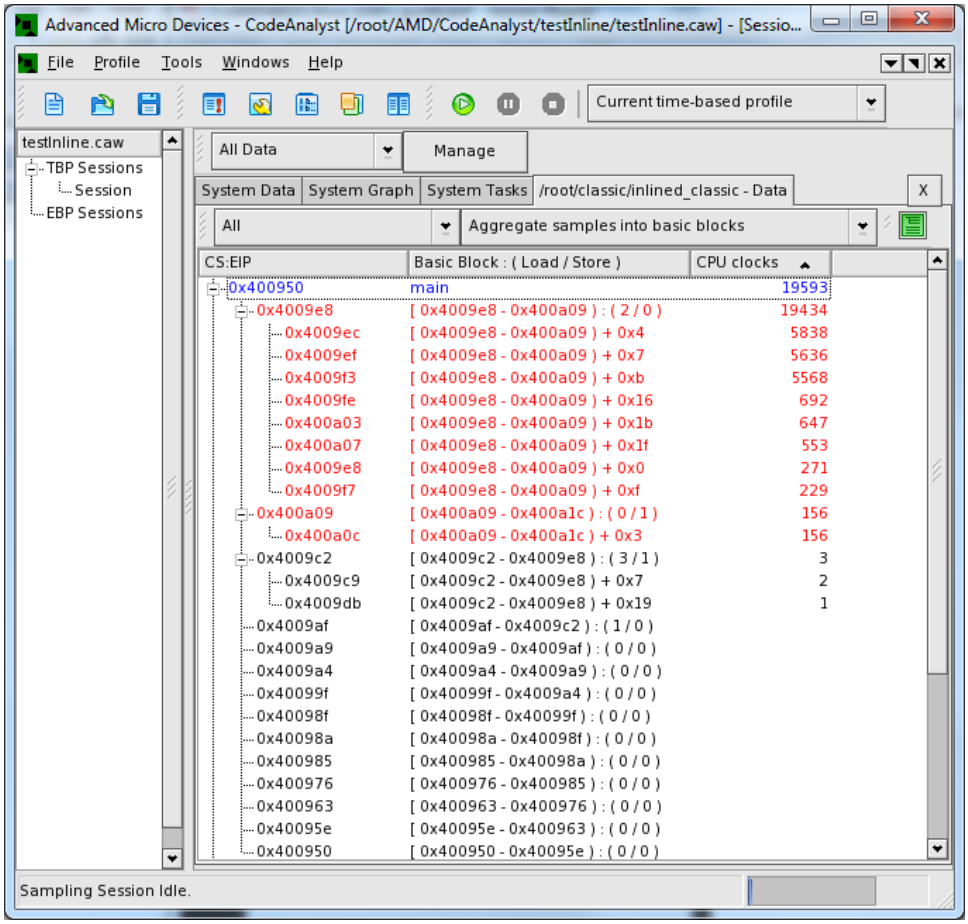
Figure 2.31. Aggregate samples into original in-line function



- Aggregate samples into basic blocks

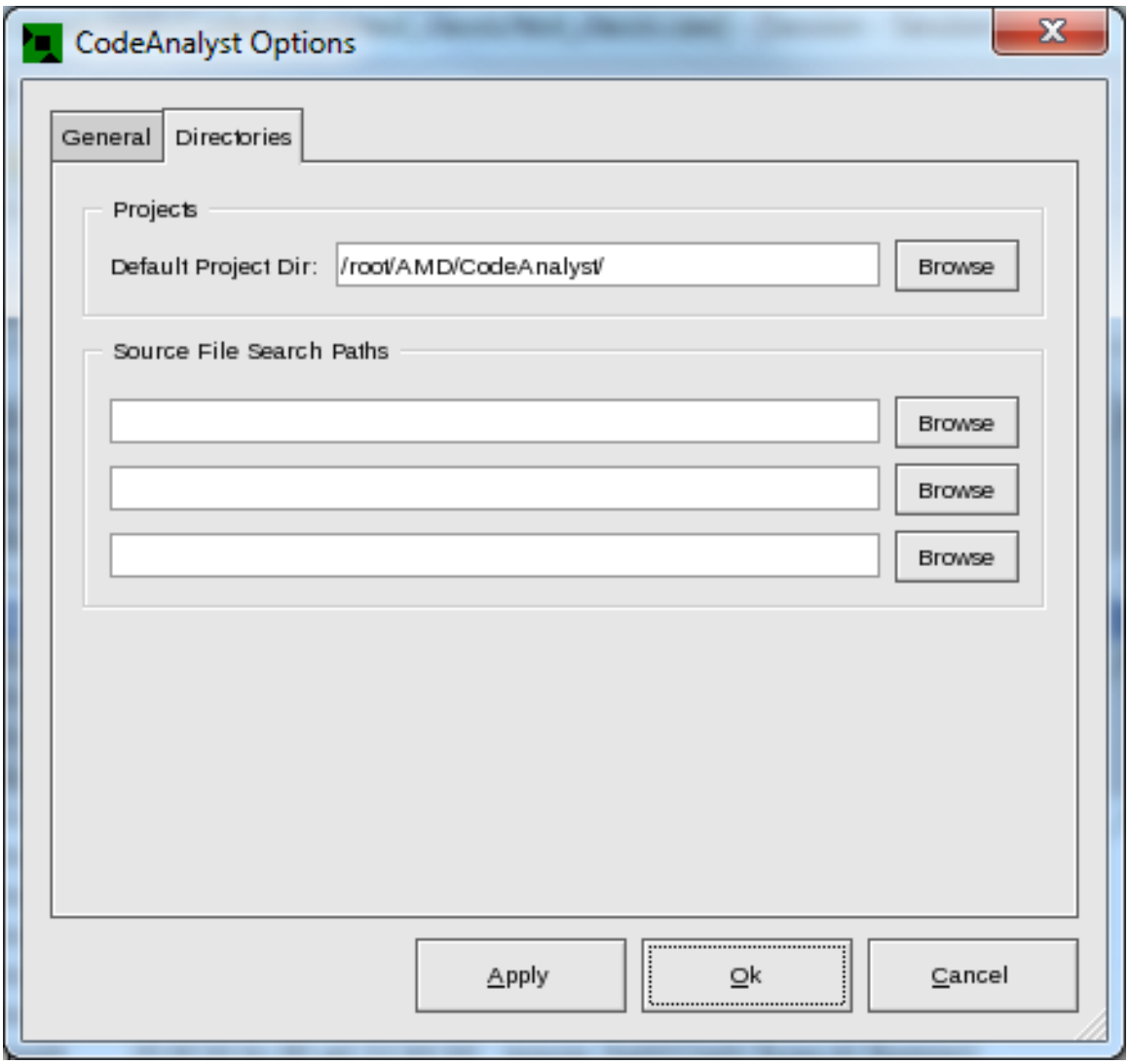
  This mode of aggregation is designed to aid basic block analysis (BBA). CodeAnalyst examines each function to identify basic blocks and aggregates samples accordingly. Each basic block is denoted by range of address as following: **[StartAddr, StopAddr) : (Number of load / Number of store)**.

Figure 2.32. Aggregate samples into basic blocks



2.3.2. Directories Tab

This tab allows the specification of directory paths to help CodeAnalyst find the information that it needs for analysis. An additional search path for finding source can be specified in the **Source File Search Paths** field.



2.4. Event Counter Multiplexing

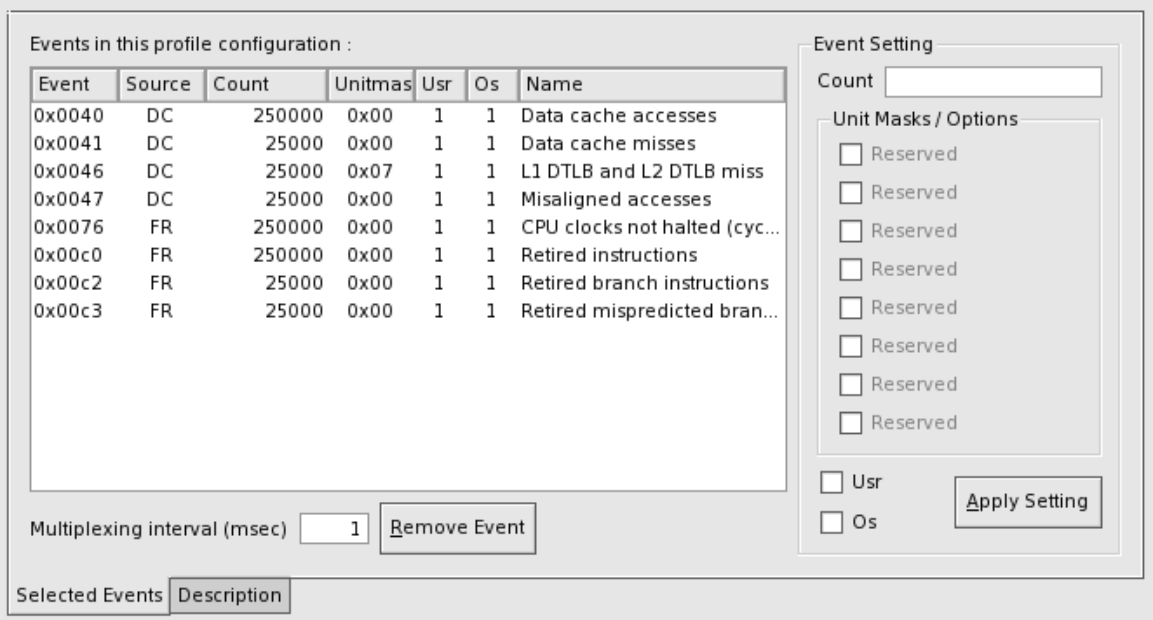
The number of performance counters in AMD processors is often limited to a small number (i.e. 4 counters in most of processor families). The number of performance events allowed in a profiling session is limited by this hardware constrain. For instance, only four events could be measured per run. A minimum of two runs was needed to collect all data for five or more events. Event counter multiplexing removes this burden.

Event multiplexing is accomplished by re-programming the performance counters with the next set of events when a timer interrups is generated by the driver. The interval between each timer can be specified in the "Edit EBS/IBS Profile Configuration" dialog box. (See Section 4.3, “Edit Event-based and Instruction-based Sampling Configuration” for more information)

2.4.1. Example of Event Counter Multiplexing

In the Edit Event Configuration dialog (see Section 4.3, “Edit Event-based and Instruction-based Sampling Configuration”), select the predefined profile configuration **"Assess performance"**.

Figure 2.33. Assess Performance Configuration (with 1 msec MUX Interval)



This profile configuration uses event counter multiplexing to measure eight performance events with one millisecond multiplexing interval. Let us assuming the number of performance counters on the running processor is four. When data collection is started with this event configuration, CodeAnalyst separates the events into two groups - (Group A: 0x40, 0x41, 0x46, 0x47) and (Group B: 0x76, 0xc0, 0xc2, 0xc3). In this scenario, CodeAnalyst samples events in Group A for the one millisecond before reprograming the hardware to samples events in Group B for the same duration. This process repeats and continues until the sampling session ends according to the run control criteria set in the "Session Settings" dialog box.

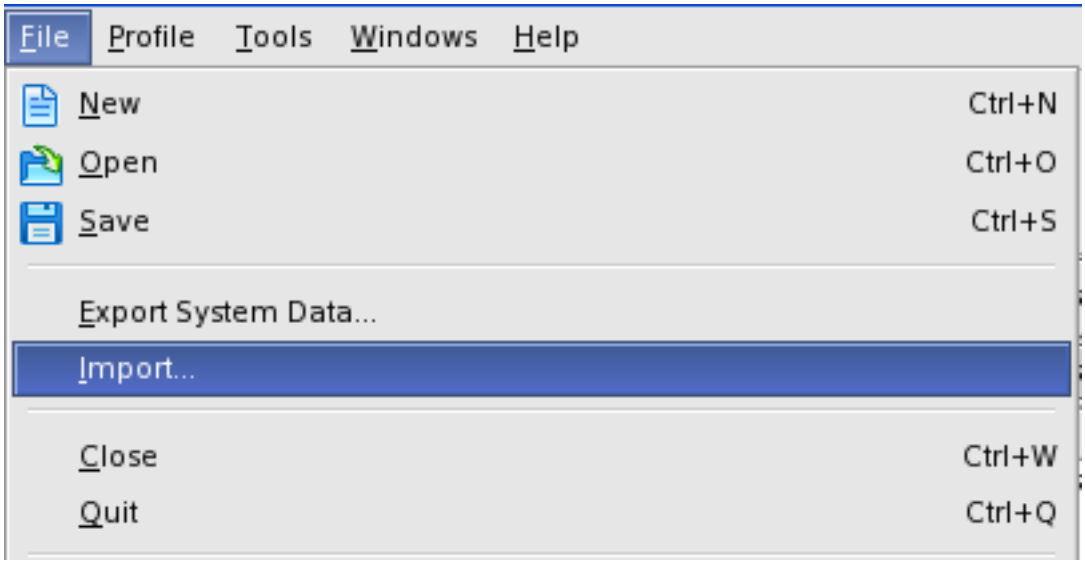
Please ensure run time is long enough to build up a statistically accurate picture of program behavior. Lengthening the duration of the data collection period may be necessary to increase the number of samples taken.

2.5. Importing Profile Data into CodeAnalyst

AMD CodeAnalyst can import profile data into a project. Typically, this features is used when profile data is collected using the Oprofile command line utility and you would like to review and analyse profile data through the GUI. Data may be from a time-based profiling, event-based profiling, Instruction-Based Sampling session, EBP/TBP files, or opreport XML file. The data to be imported must be generated by Oprofile command line tool. A new session is created for the data. This section illustrates the process of importing.

1. Collect profile data using the Oprofile. Please refer to Oprofile documentation [http://oprofile.sourceforge.net/docs] on how to collect profile data. After profiling session, profile data is usually stored in /var/lib/oprofile/samples/current/ directory.

2. Select the **"Import..."** menu item from the **"File"** menu.



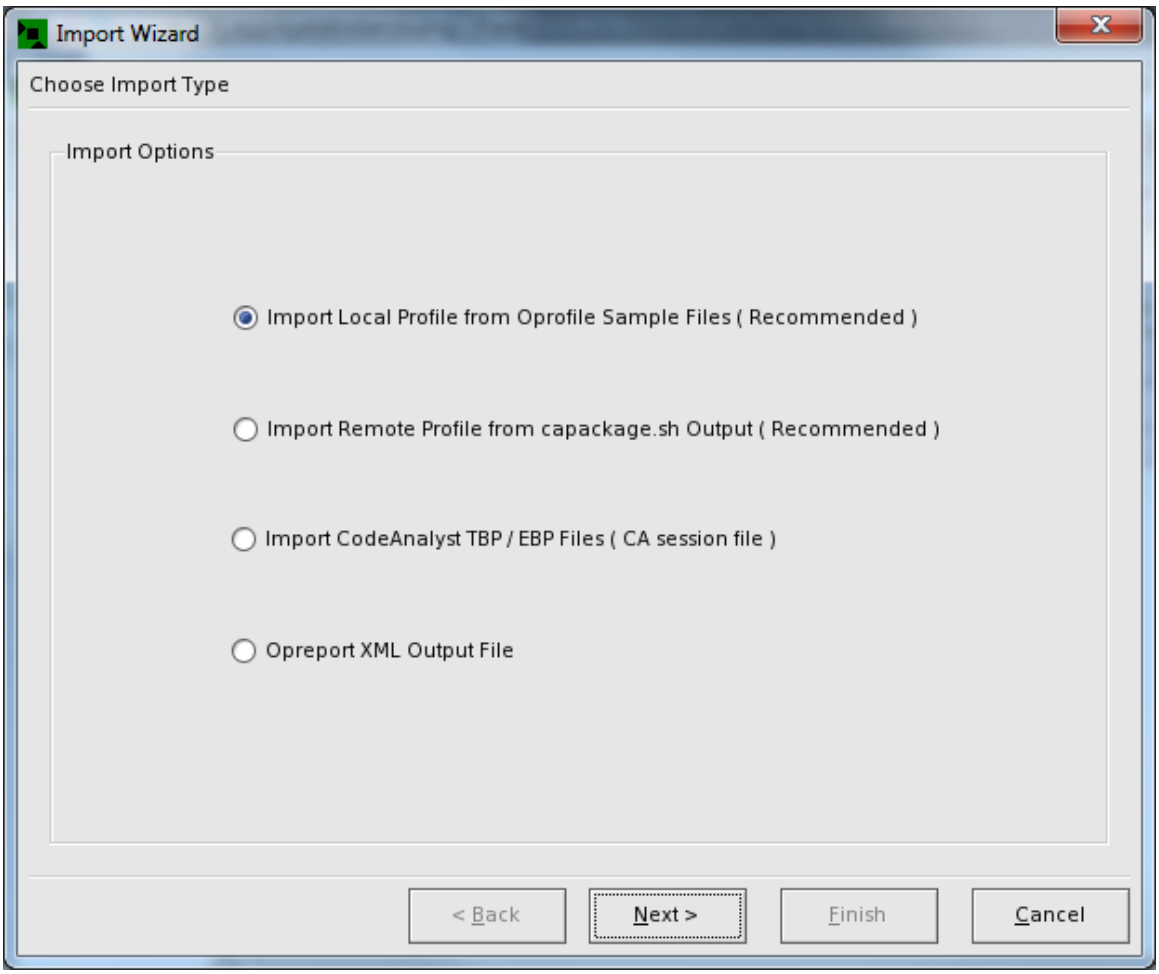
3. An Import Wizard dialog box appears. Codeanalyst can import four types of profile data:

   - **Remote Profiling:** In this mode, the profile data from a remote system can be packaged into a compressed tarball which is generated by a script called **capackage.sh**. Then, CodeAnalyst can import the packaged file for analysis. (Advande User only)

   - **Local Profile:** In this mode, the profile data is generated on the local machine.

   - **TBP/EBP File:** TBP/EBP files store profile data for each CodeAnalyst session. This can be import into different CodeAnalyst project.

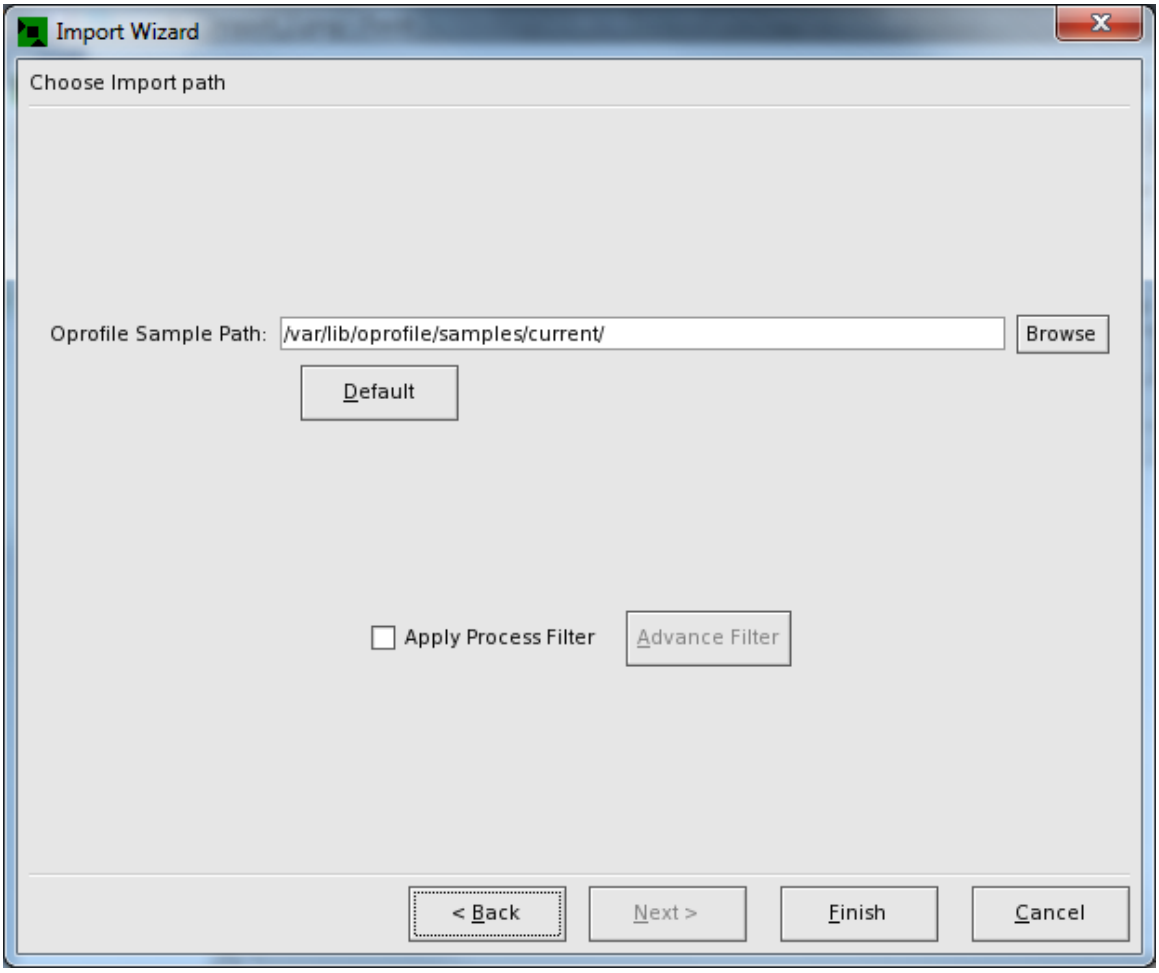
   - **Opreport's XML File:** Opreport is an OProfile's commandline utility for viewing profile data in tabulate style. It can also export data into an XML file.



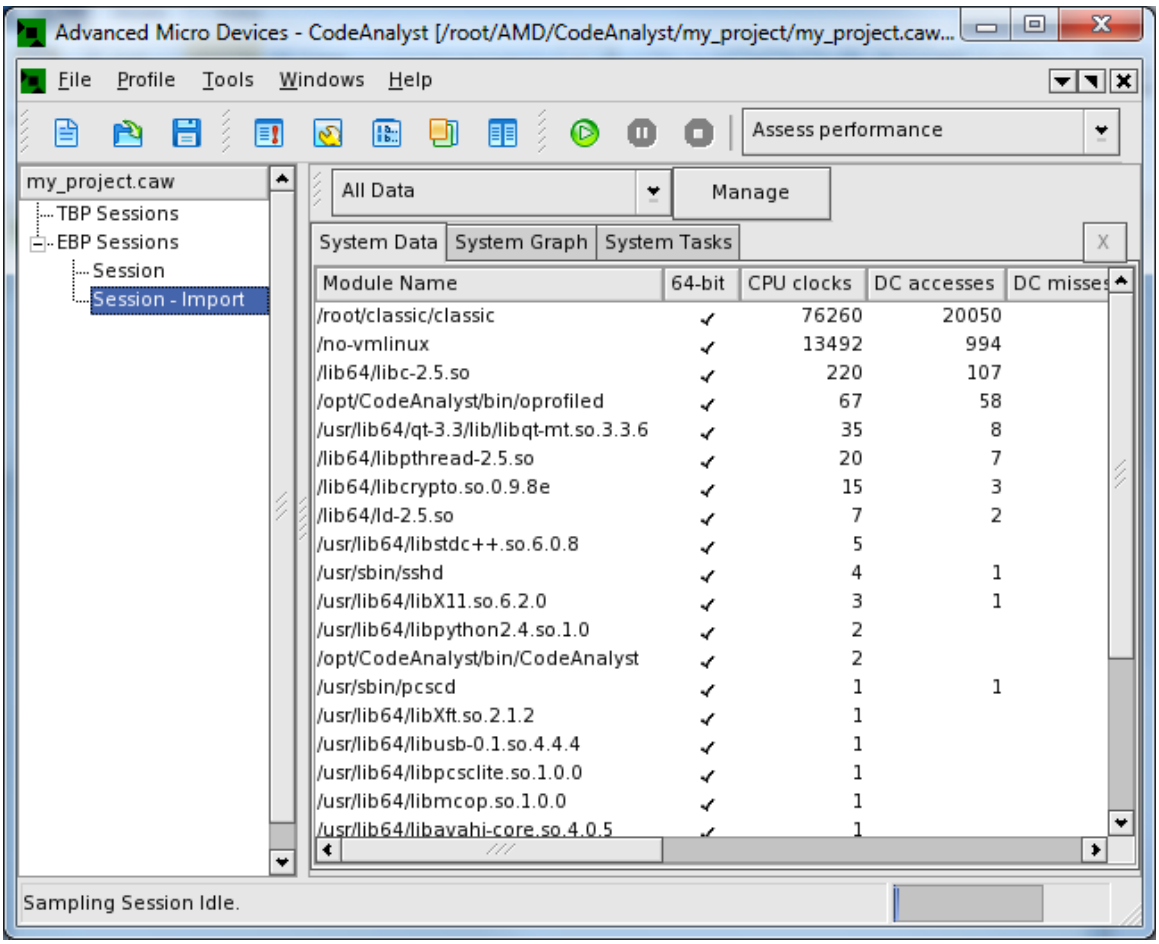
2.5.1. Import Local Profiling

If you choose **Local Profiling** and click **Next**, the wizard will prompt you to enter the location where profiling data is stored. The default location is **/var/lib/oprofile/samples/current/**. Then click **Finish**.

Figure 2.34. Import Wizard

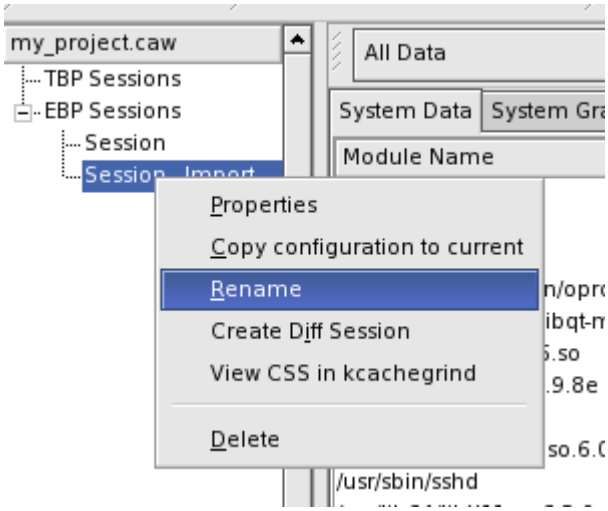


CodeAnalyst creates a new session for the imported data (**Session import**). The GUI displays the imported data in the System Data, System Graph and System Tasks tabs. Users can also choose to import profile data of any particular processes and the dependent modules by specifying the full-path of each binary and using **Advance Filter**.

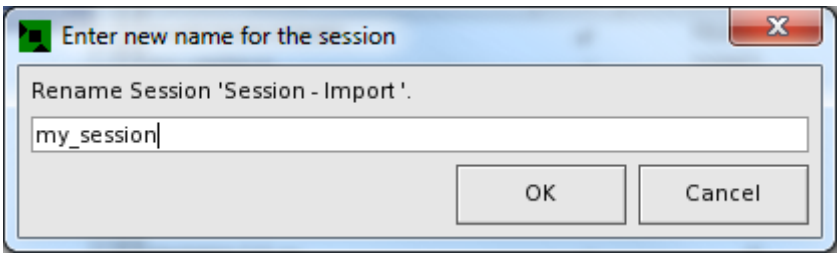


1. Right click on the session name for the imported data.

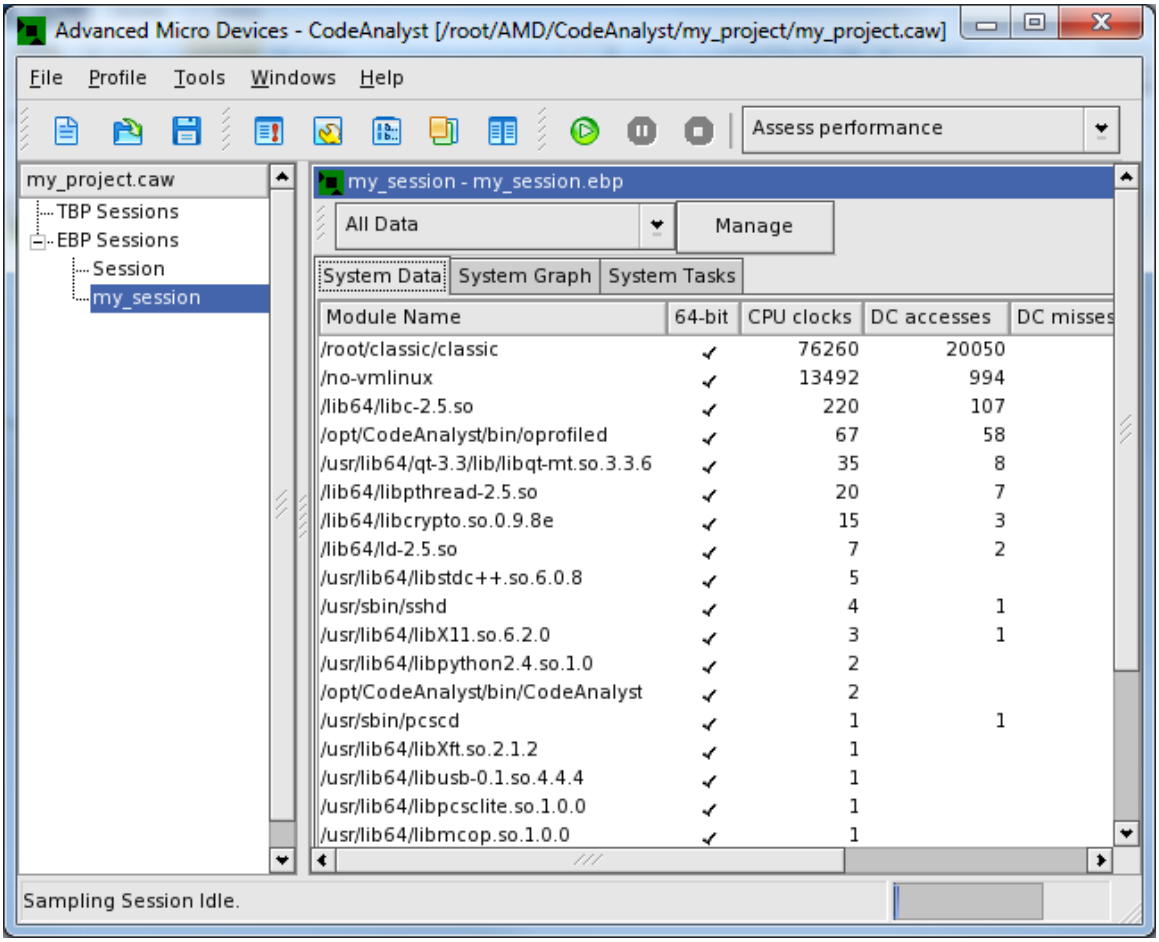
2. CodeAnalyst displays a pop-up, contextual menu. Select **Rename** from the pop-up menu to rename the session.



3. A dialog box appears asking for the new session name. Enter the new session name in the text field.



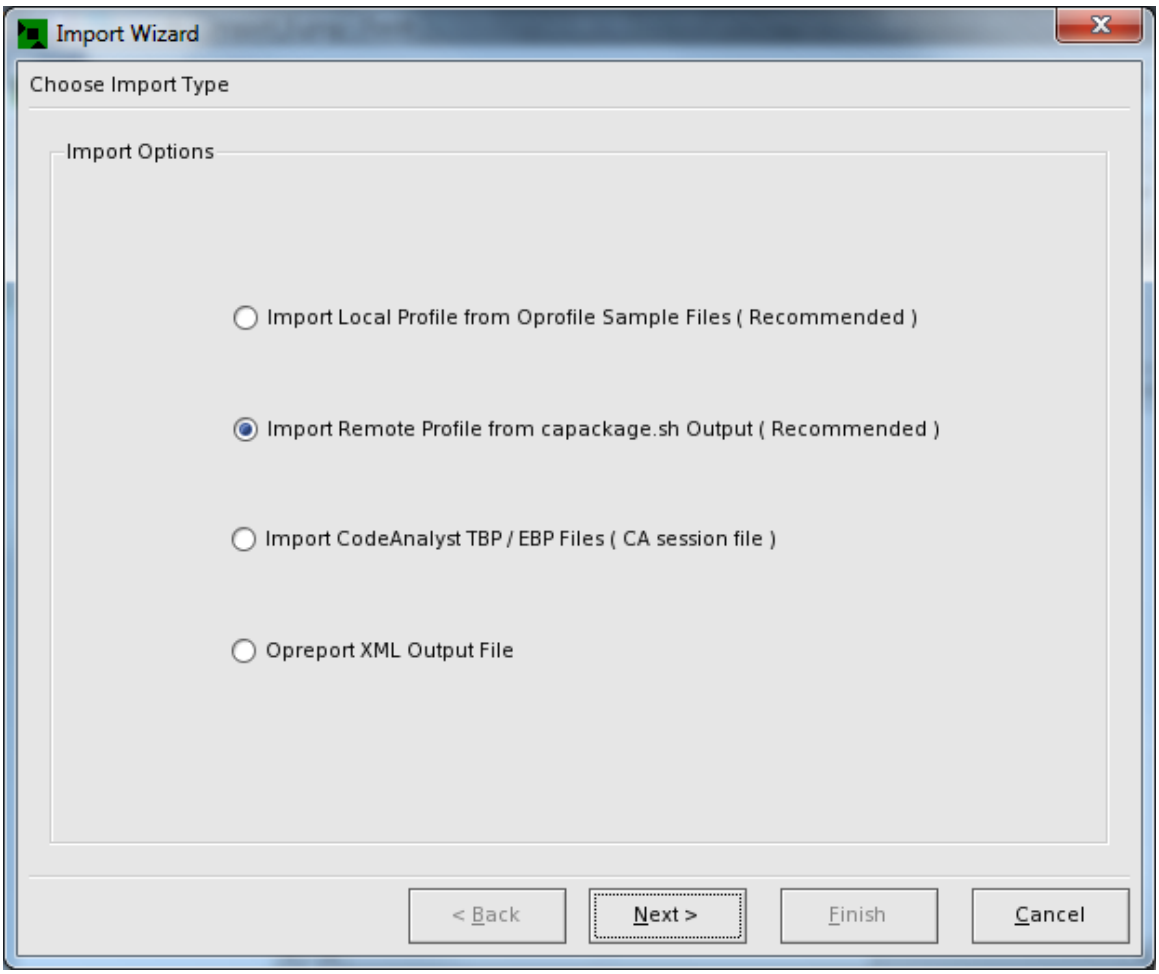
CodeAnalyst changes the name of the session in the session management area of the workspace.



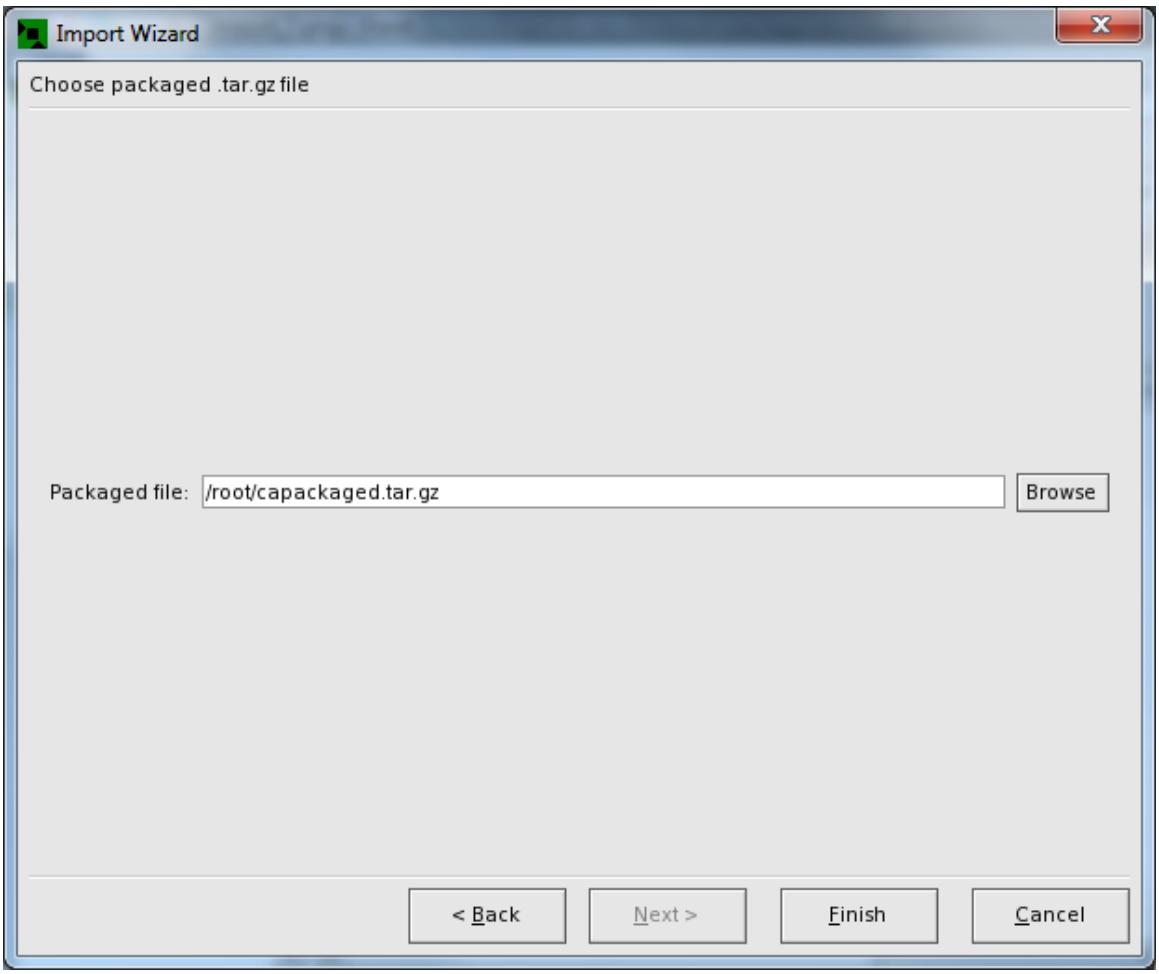
2.5.2. Import Remote Profiling

When graphical-user-interface is not available on the system, users can generate profile using **"op-control"** command-line tool(Please see Section 2.8, “CodeAnalyst and OProfile”). Once finished, the profiling information can be transferred to another system and viewed by CodeAnalyst GUI. CodeAnalyst provides a tool called **capackage.sh**. It gathers information necessary for analyzing a session of profiling, and compress them into an easily managed tarball (**capacked.tar.gz**) This tarball can then be transfer onto another system and imported into CodeAnalyst GUI.

If you choose **Remote Profiling** and click **Next**,



The wizard prompts the user to enter the location of the tarball **capacked.tar.gz** output from **capackage.sh**.



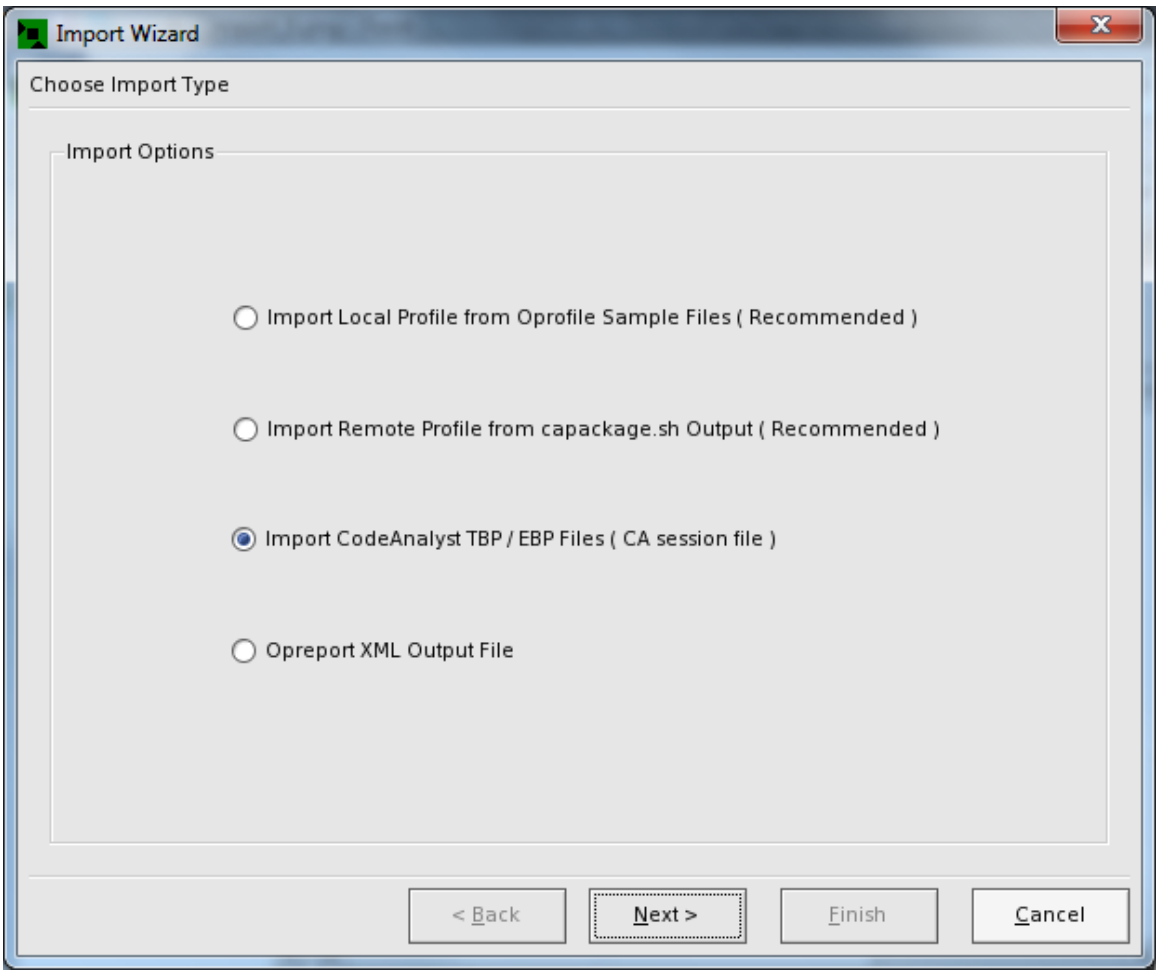
Once completed, click **Finish** and CodeAnalyst will untar **capacked.tar.gz** into the **/tmp/CAxxxxxx/capacked/** directory, which contains the following sub-directories:

- **binary**—Stores the executable and modules used in profiling.
- **current**—StoresOprofile samples.
- **Java**—Stores information related to Java profiling.

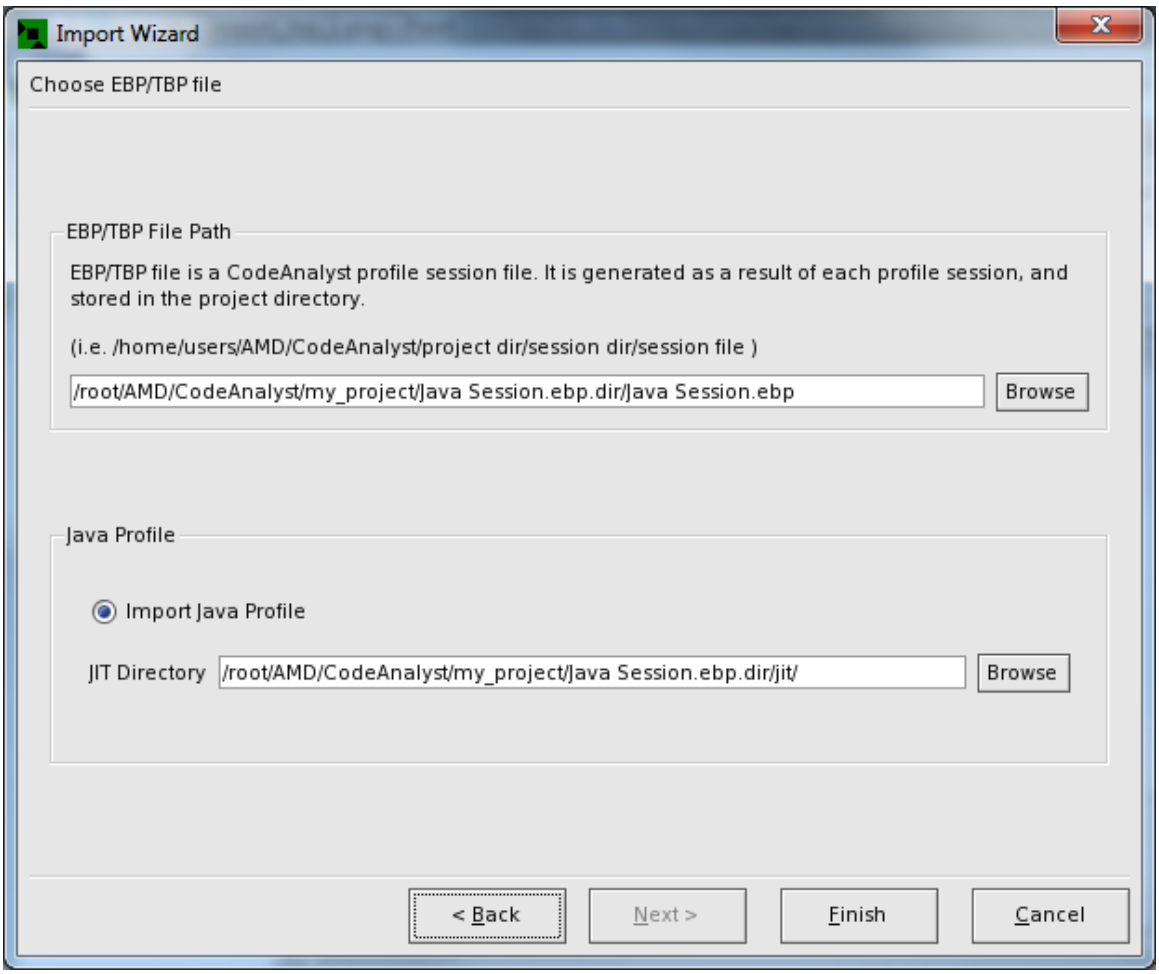
Then the importing process continues as described in Section 2.5.1, “ Import Local Profiling”.

2.5.3. Import TBP / EBP Files

TBP and EBP files are generated by CodeAnalyst to store profiling data of each session. Importing profile sessions by TBP/EBP files allows users to easily move around profiling session onto different system for viewing because the file is relatively much smaller than the **capackage.tar.gz** file required by normal remote profiling. In this case, data is already processed and does not require an additional data processing step, which can take several minutes. However, session properties are not available in this case.



TBP and EBP files are generally stored in the project directory. When using the **Import Java Profile** session, the full-path to the JIT directory must also be specified in the **JIT Directory** field.



2.5.4. Import Opreport's XML Output Files

2.6. Exporting Profile Data from CodeAnalyst

AMD CodeAnalyst can export profile data from a table (such as the System Data tab or Processes tab) or source view. The data is exported as a file containing comma separated values (CSV.) This section illustrates the process of exporting data to a .CSV file.

1. With an open project and session, click the **System Data** tab and select the **Export System Data...** item from the **File** menu.

2. With an open project and session, select the **Export System Data...** item from the **File** menu.

3. Select or enter the name of the .CSV file to which the data from the System Data tab is to be written.

4. Click the **Save** button. The data is converted to CSV format and is written to the file.

5. Launch a spreadsheet program like Microsoft Excel or OpenOffice.org Calc.

6. Import the .CSV file into the spreadsheet.

Figure 2.35. Select "Export System Data"

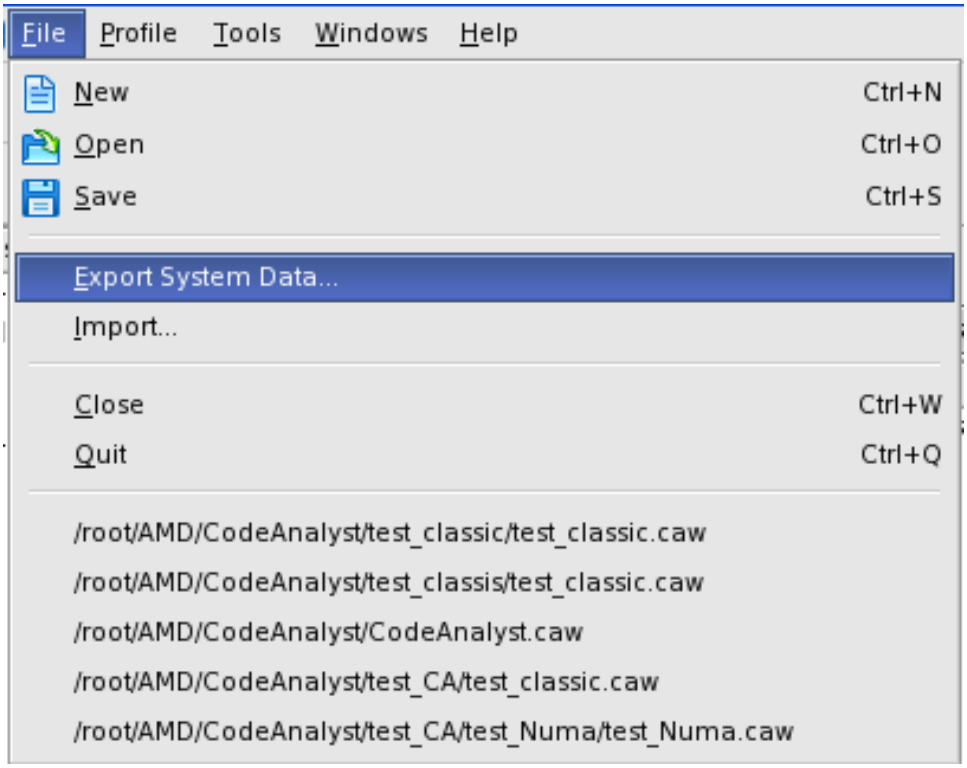


Figure 2.36. Specify output CSV file

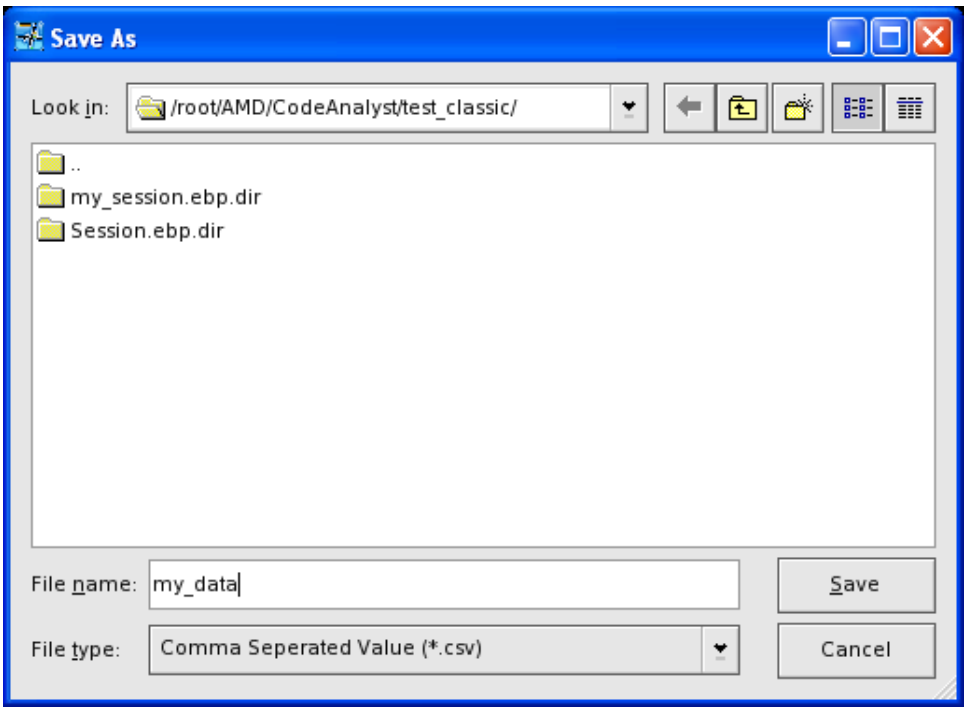
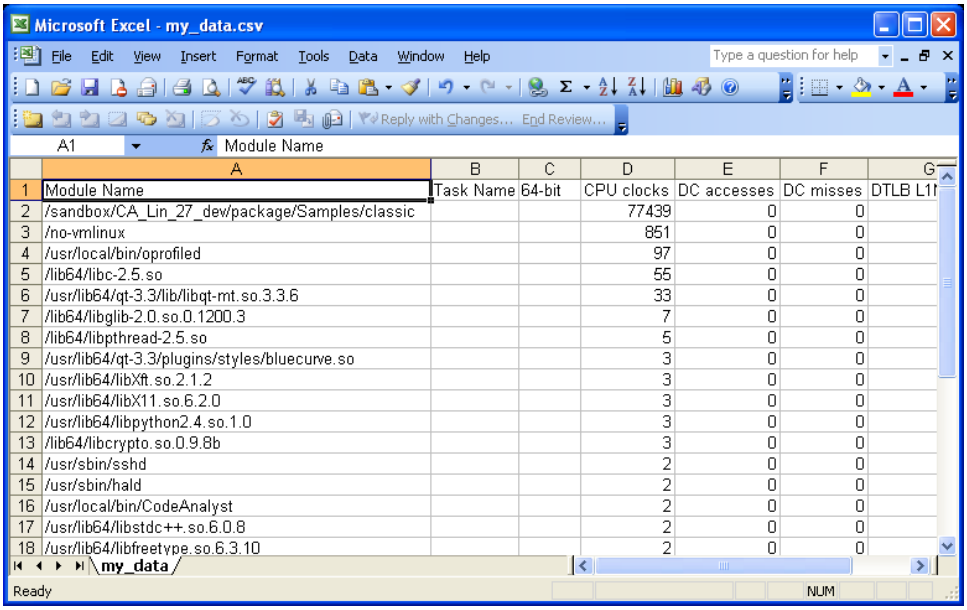


Figure 2.37. Import CSV into a spreadsheet

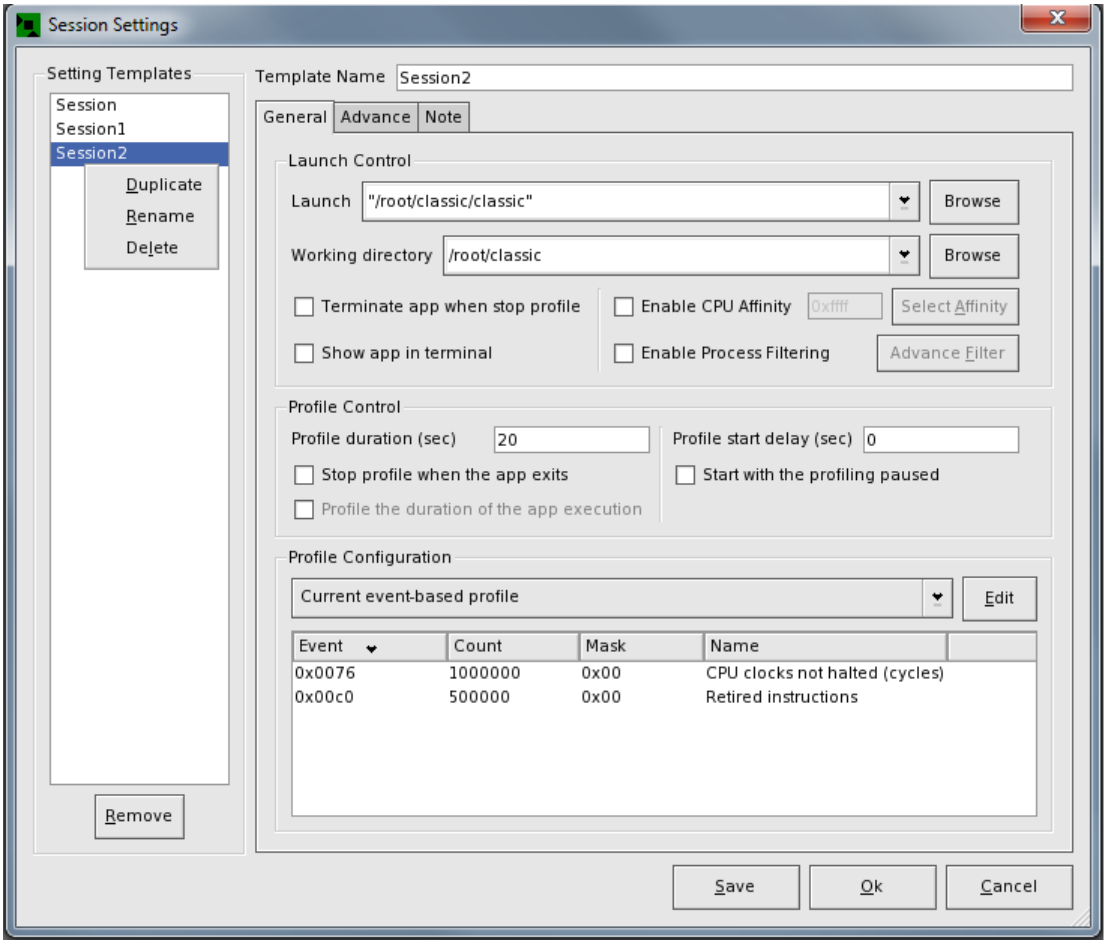


	A	B	C	D	E	F	G
1	Module Name	Task Name	64-bit	CPU clocks	DC accesses	DC misses	DTLB L1f
2	/sandbox/CA_Lin_27_dev/package/Samples/classic			77439	0	0	
3	/no-vmlinux			851	0	0	
4	/usr/local/bin/oprofiled			97	0	0	
5	/lib64/libc-2.5.so			55	0	0	
6	/usr/lib64/qt-3.3/lib/libqt-mt.so.3.3.6			33	0	0	
7	/lib64/libglib-2.0.so.0.1200.3			7	0	0	
8	/lib64/libpthread-2.5.so			5	0	0	
9	/usr/lib64/qt-3.3/plugins/styles/bluecurve.so			3	0	0	
10	/usr/lib64/libXft.so.2.1.2			3	0	0	
11	/usr/lib64/libX11.so.6.2.0			3	0	0	
12	/usr/lib64/libpython2.4.so.1.0			3	0	0	
13	/lib64/libcrypto.so.0.9.8b			3	0	0	
14	/usr/sbin/sshd			2	0	0	
15	/usr/sbin/hald			2	0	0	
16	/usr/local/bin/CodeAnalyst			2	0	0	
17	/usr/lib64/libstdc++.so.6.0.8			2	0	0	
18	/usr/lib64/libfreetype.so.6.3.10			2	0	0	

2.7. Session Settings

The **"Session Settings"** specify information that is needed to control performance data collection. Session Settings are persistent and apply to future data collection sessions that are initiated within a project until the Session Settings are again changed.

Figure 2.38. Session Settings Dialog



2.7.1. Setting Templates

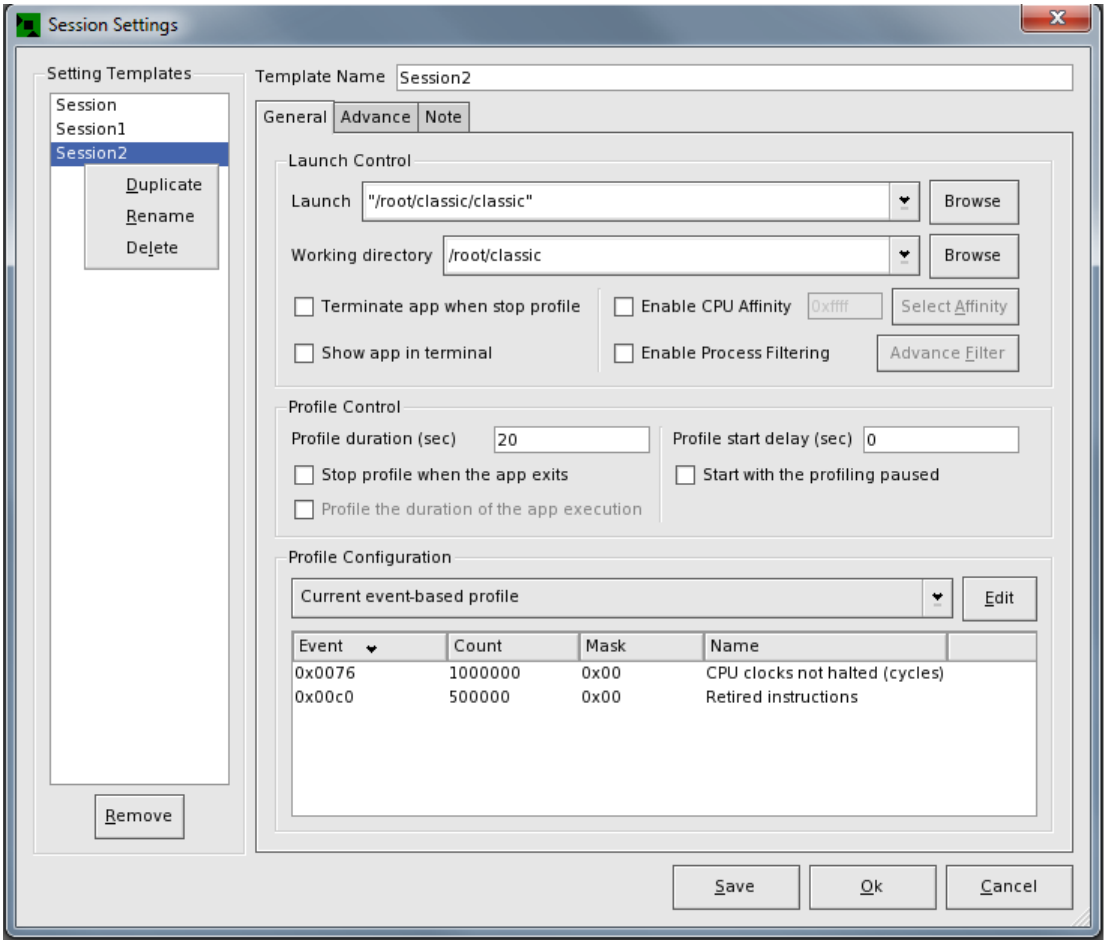
Once users finish configuring a data collection session, settings can be stored as a "setting template". The currently selected template will be used to configure future data collection sessions. Stored templates are listed in the "Setting Templates" field. When template selection changes, the dialog will re-populate each field with the settings previously stored. The template provides convenience when performing multiple data collection with different settings.

Right click on the selected template to show options to "Rename", create "Duplicate", or "Delete" each template. Renaming the current template can also be done simply by modifying the "Template Name" field and click "Save" or "Ok" button. Click the "Remove" button to remove the currently selected template.

NOTE that the modified template must be saved by clicking the "Save" or "Ok" button before selecting another template.

2.7.2. General Tab

Figure 2.39. Session Settings Dialog: General Tab



2.7.2.1. Template Name

This is a name that is assigned to the session. If the session name is not changed, CodeAnalyst will auto-generate new session names by appending a number to the end of the base session name.

2.7.2.2. Launch Control

2.7.2.2.1. Launch and Working directory

- Launch - Users can specify the application program to launch in the Launch field. Enter the path to the executable program to be launched. You may also enter the path to a shell script file to be started instead of an executable program. You may also leave this field blank in order to perform system-wide data collection when overall system monitoring is required.
- Working Directory - is the working directory for the application to be launched. Enter the path to the working directory in this field.

In addition to directly entering path names into the "Launch" and "Working directory" fields, you may browse to the desired location by clicking the appropriate "Browse" button. Each field also offers a drop-down list of the most recently used path names. The drop-down lists retain the last 10 application paths and the last 10 working directory paths, respectively.

2.7.2.2.2. Options

- **Terminate the app after the profile** - terminates the application at the completion of the profile sampling duration.

- **Show app in terminal** - Run the target application in a terminal. This options allows users to access stdin/stdout/stderr from the command-line.

- **Enable CPU Affinity** - specifies the list of CPUs allowed to run the target application. (See Section 2.7.6, “Changing the CPU Affinity”)

- **Enable Process Filter** - filters out processes during data processing unless specified in "**Advance Filter**". (See Section 2.7.7, “Process Filter”)

2.7.2.3. Profile Control

- **Stop data collection when the app exits** - terminates the sampling process if the application terminates. This option is convenient when profiling small applications or if the shutdown sequence is desired in the profile. Selecting this option enables the **(s) Profile Duration** option that sets up the profile run-time in seconds. (Profiling large applications over a long time period with this option could create very large profile data files.)

- **Profile the duration of the app execution** - allows profiling to continue as long as the specified application is running. When this option is selected, the (s) Profile duration option is disabled and no specific profiling time limit is needed.

- **Profile duration (sec)** - sets the profile sampling duration in seconds.

- **Start with profiling paused** - This option is included for times when using the profiler API to programmatically control the Pause and Resume functionality.

- **Start delay (sec)** - sets the time in seconds of delaying the profile sampling after the target application is launched.

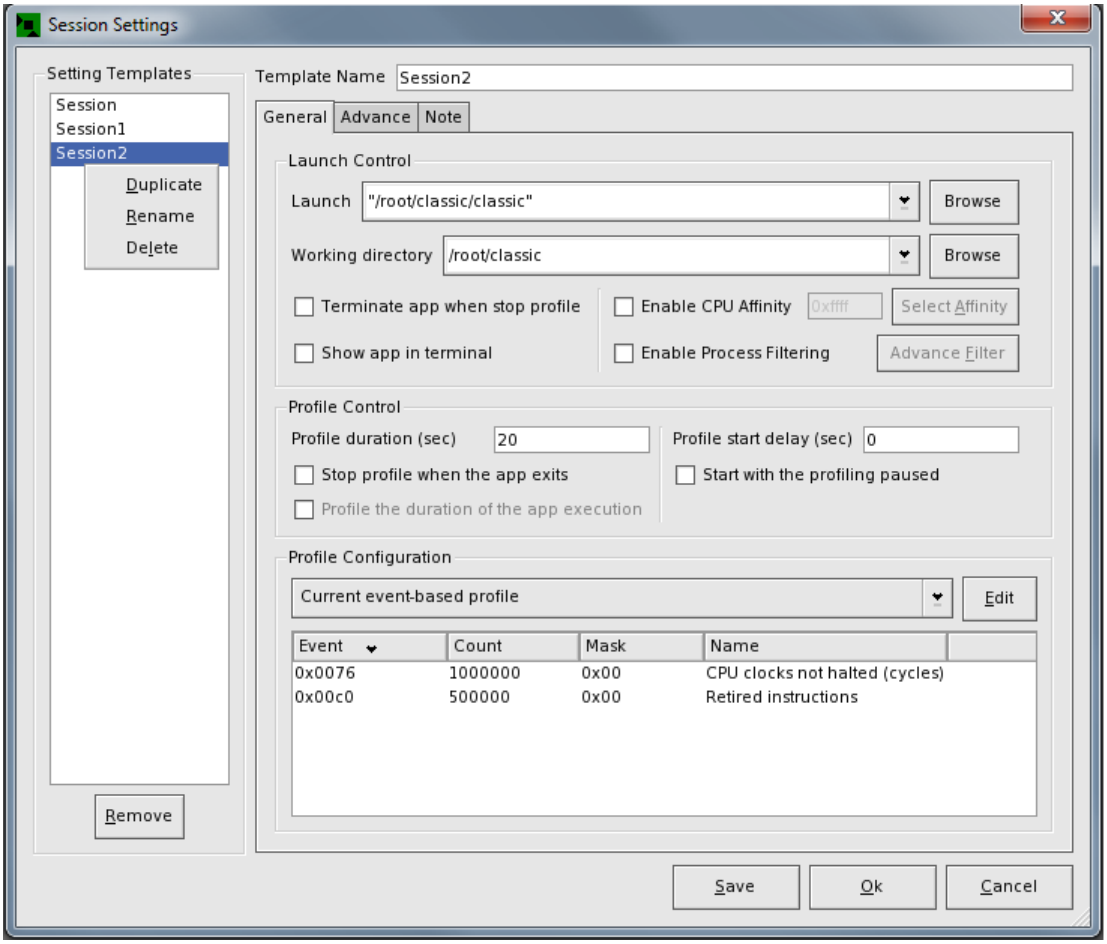
2.7.2.4. Profile Configuration

This is the kind of analysis to be performed. Choose a predefined profile configuration from the dropdown list. After selecting a profile configuration, performance events along with the count and unitmask settings will be shown in the list. Please see Chapter 4, *Configure Profile* and Section 4.5, “Manage Profile Configurations” for more detail.

Click the "**Edit**" button to edit the selected profile configuration. However, predefined profile configurations cannot be changed unless saved as a new name.

2.7.3. Advance Tab

Figure 2.40. Session Settings Dialog: Advance Tab



2.7.3.1. Enable vmlinux

This settting allows users to specify the kernel image used for kernel and kernel modules profiling. Uncheck this setting to profile without the kernel image. The option equals "no-vmlinux" option in Oprofile. Check this setting to specify the vmlinux file. If vmlinux is compressed, uncompress it first. Please note that this is not "vmlinuz" file.

2.7.3.2. OProfiled Buffer Configuration

- Event Buffer Watershed Size - Set kernel buffer watershed to num samples (2.6 only). When it'll remain only buffer-size - buffer-watershed free entry in the kernel buffer data will be flushed to daemon, most useful values are in the range [0.25 - 0.5] * buffer-size.

- Event Buffer Size - Number of samples in kernel buffer. When using a 2.6 kernel buffer watershed needs to be tweaked when changing this value.

- CPU Buffer Size - Number of samples in kernel per-cpu buffer (2.6 only). If you profile at high rate it can help to increase this, if the log file shows excessive count of sample lost cpu buffer overflow.

2.7.3.3. Enable Call Stack Sampling (CSS)

NOTE: This feature requires a specialized OProfile daemon and OProfile kernel module not available publicly at the moment.

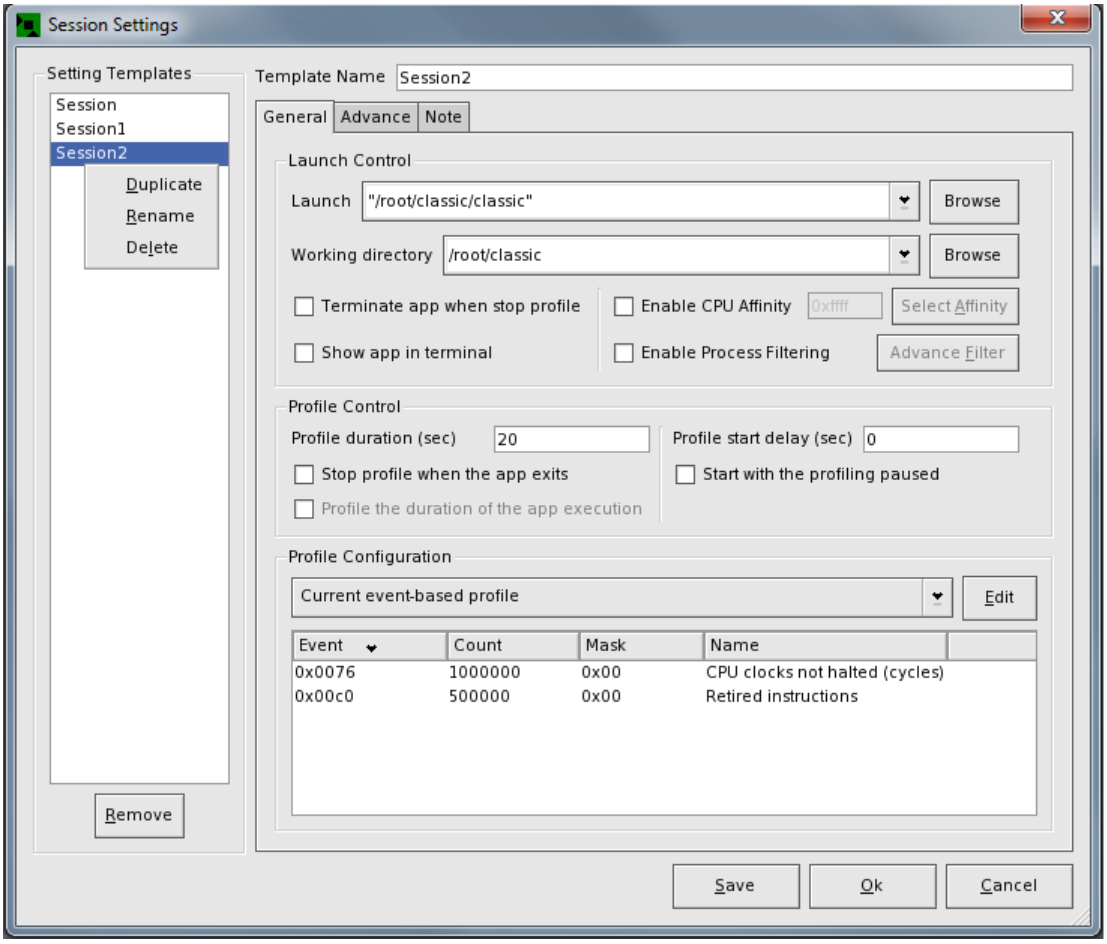
- Call Stack Depth - specify the maximum depth of call stack unwinding

- Call Stack Unwinding Interval - specify the frequency of stack unwinding (i.e. perform stack unwinding every 1000 samples.)

- TGID for CSS - Users can specify the TGID to use for CSS instead of using TGID of the launched target application. Note that if you select "Use TGID of the launched target application", the "Launch" field must be specified.

2.7.4. Note Tab

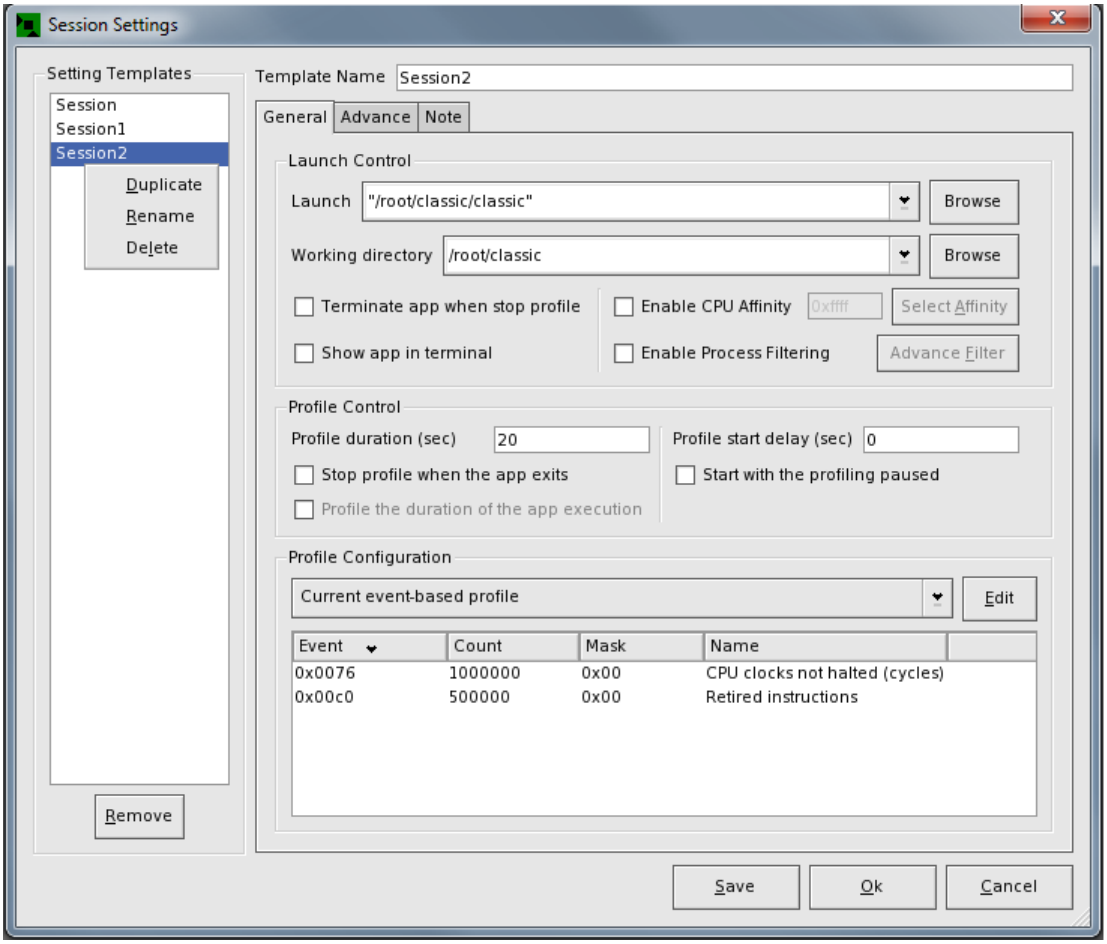
Figure 2.41. Session Settings Dialog: Note Tab



This tab contains a field where users can specify profile session note.

2.7.5. OProfiled Log

Figure 2.42. Session Settings Dialog: OProfiled Log Tab (Property Mode Only)

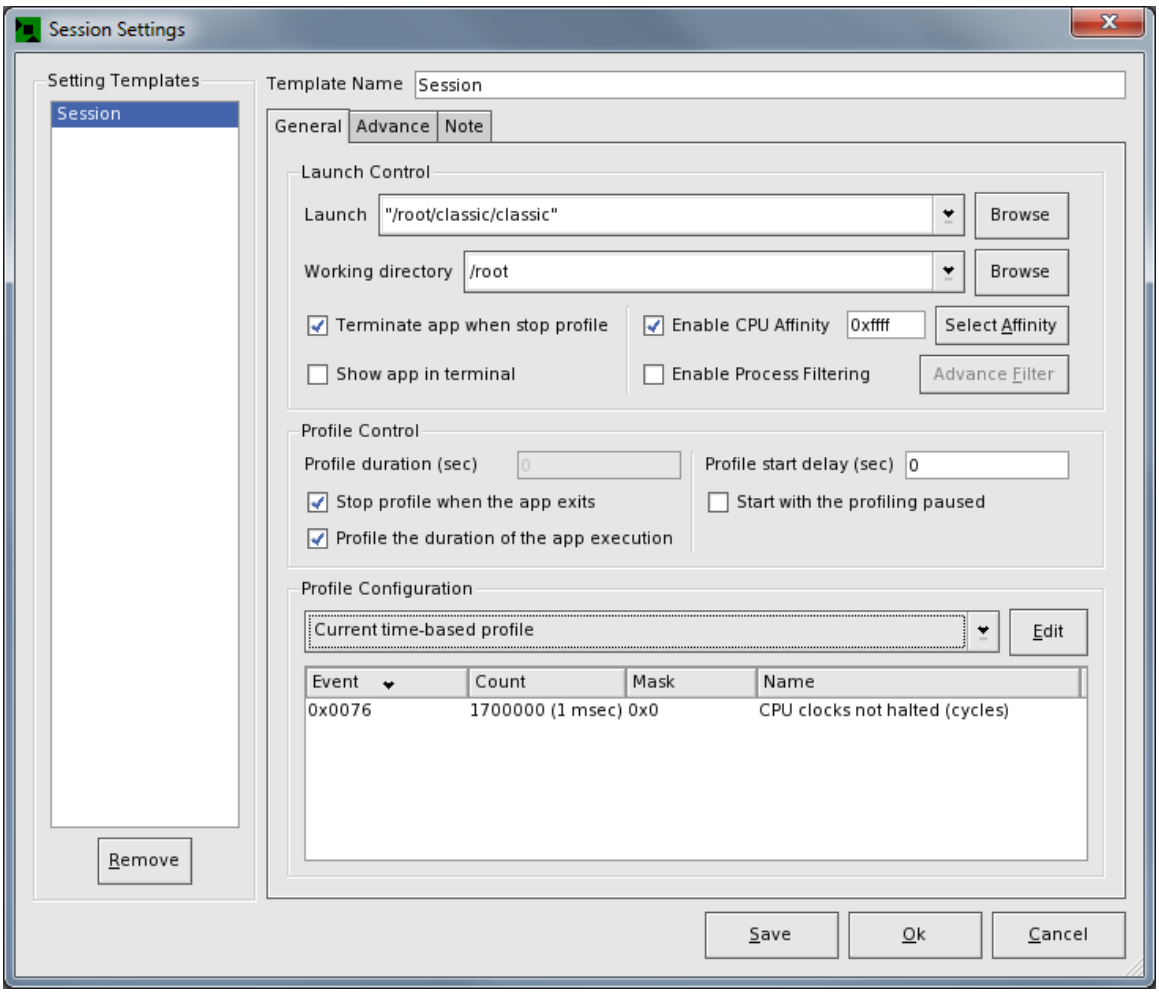


This tab contains the information from `/var/lib/oprofile/samples/oprofiled.log`. This is available only in property mode of the Session Setting dialog.

2.7.6. Changing the CPU Affinity

CPU affinity limits the execution of a program or process to selected cores in a multicore system. CPU affinity is set through a **CPU affinity mask** in which each bit of the mask specifies whether the program or process may execute upon a particular core. The number of available cores is system dependent. CPU affinity can be used to perform scalability analysis by limiting the number of cores available to a multi-threaded program.

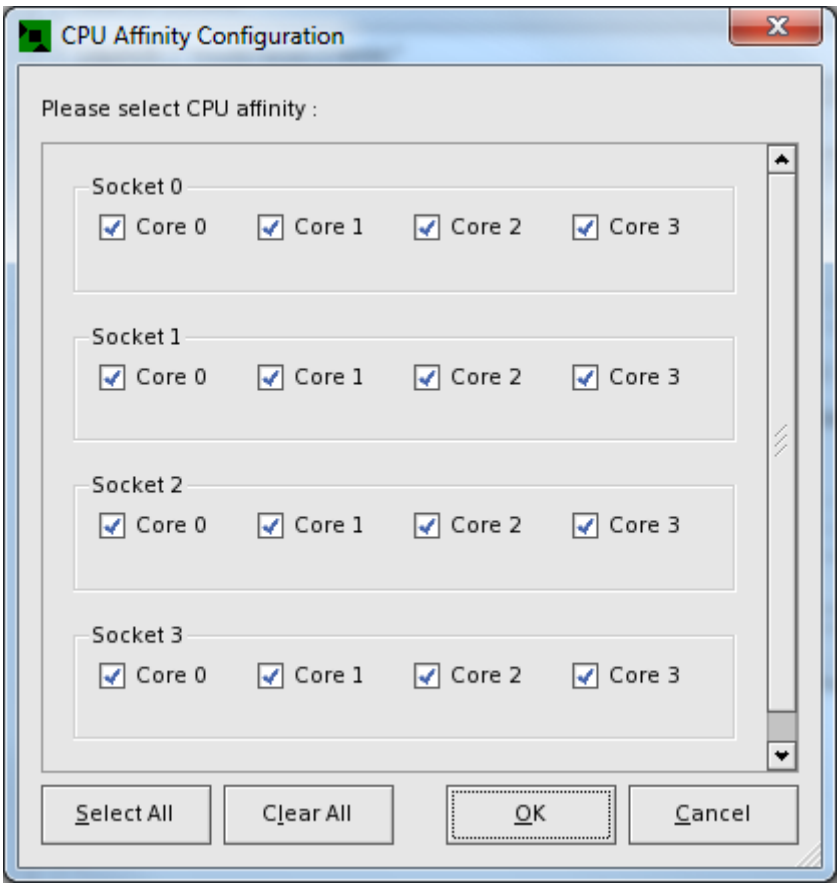
CPU affinity is defined in the CodeAnalyst **Session Settings** dialog box. The CPU affinity mask can be specified directly as a hexadecimal value in the **CPU affinity mask** field as shown in the screen shot below. The CPU affinity mask determines the CPU affinity for the application program that is launched by CodeAnalyst.



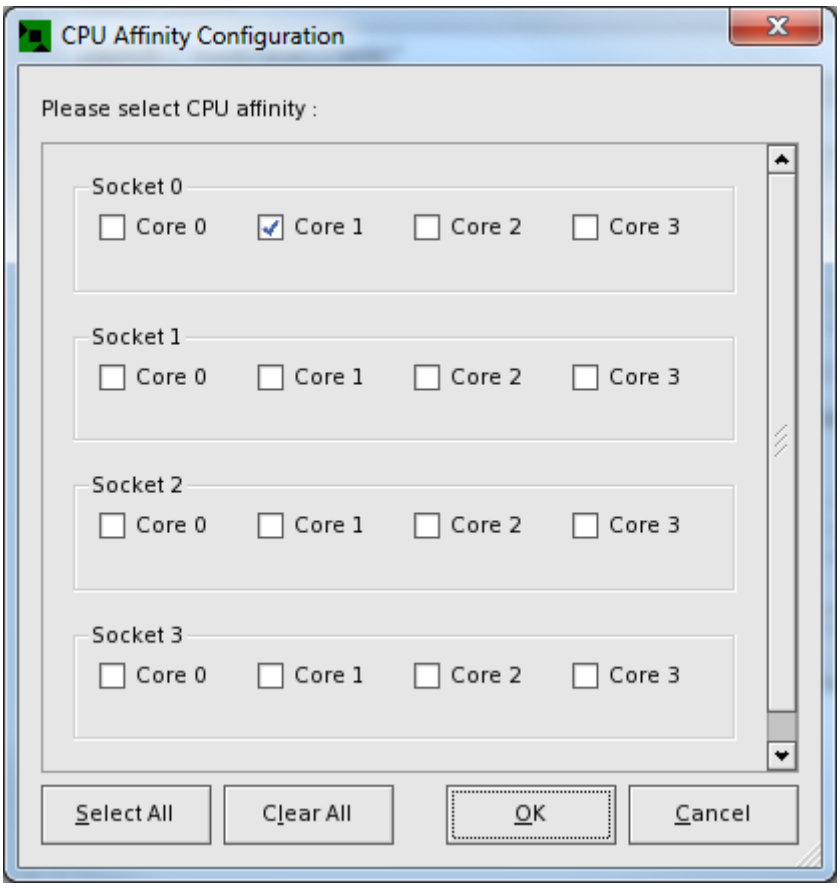
It may be more convenient to set CPU affinity using the "Select Affinity" button located to the right of the **CPU affinity mask** field.

To change the CPU affinity:

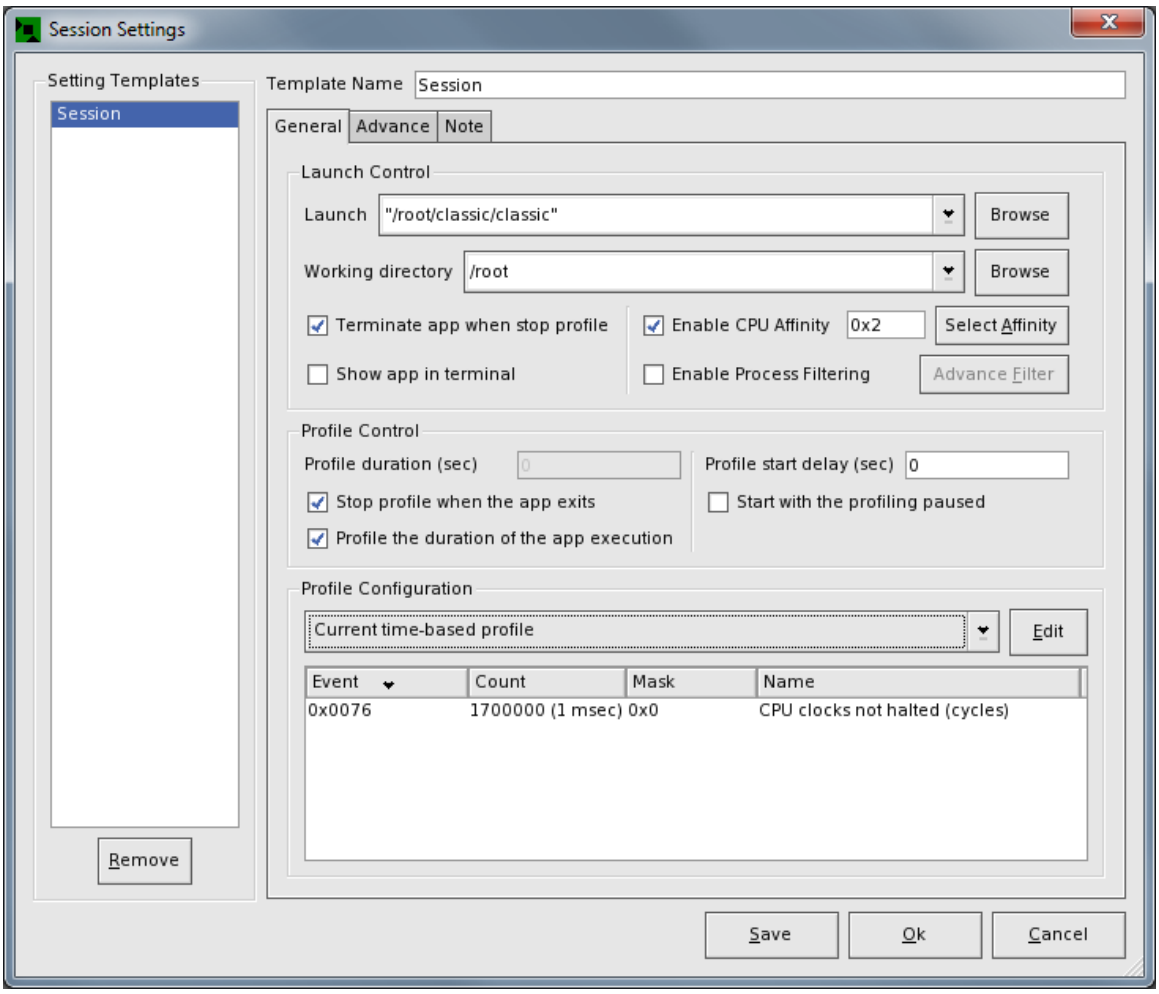
1. Open the **Session Settings** dialog box.
2. Click the "Select Affinity" button.
3. The **CPU Affinity Configuration** dialog box appears.
4. Check a box to enable execution on a core.
5. Click the **Select All** button to check all boxes, enabling execution on all cores.
6. Click the **Clear All** button remove checks from all boxes.
7. Click the **OK** button to activate the CPU affinity settings.



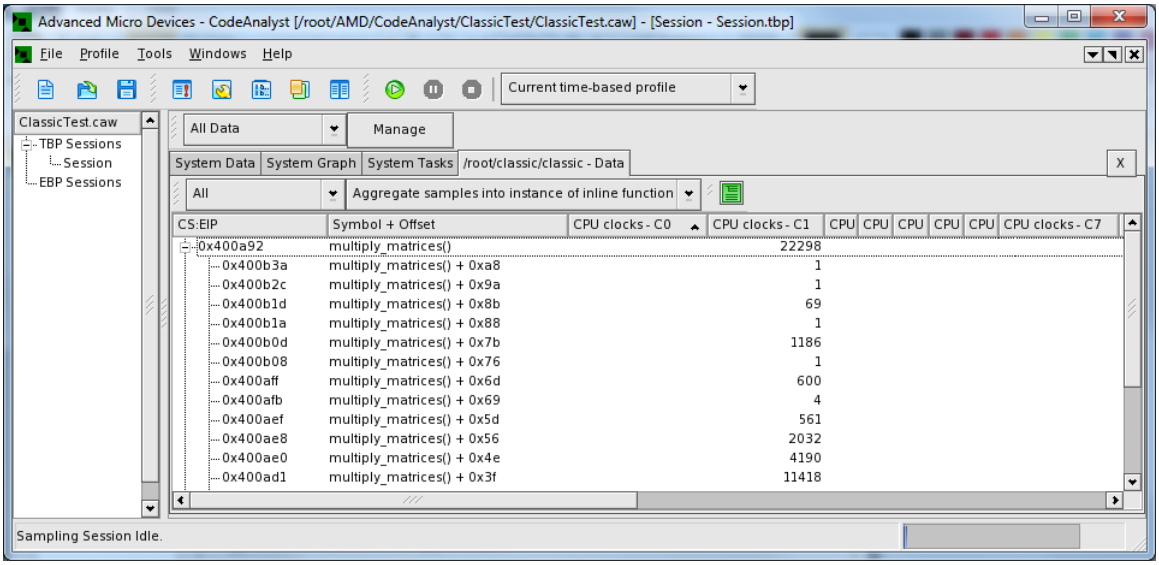
The CPU affinity configuration below limits execution of the application program under test to a single CPU (Core 1.)



The Session Settings dialog box below reflects the change to the CPU affinity mask (0x2.)



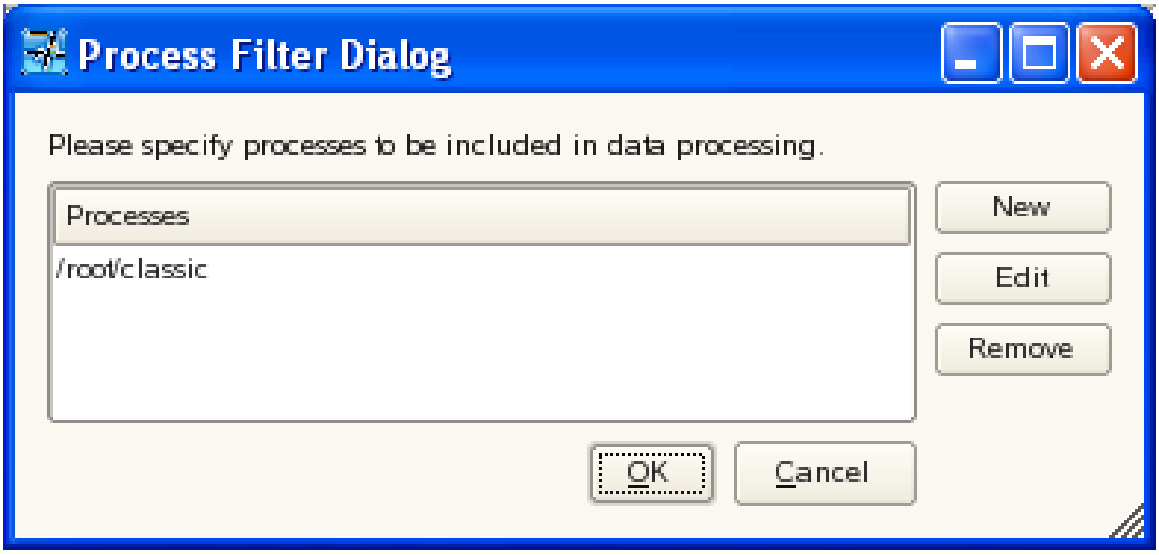
When CodeAnalyst launches the application program `/root/classic` using the CPU affinity mask, execution is restricted to core 1. The following screen shot shows that execution was indeed limited to core 1 since all timer samples for classic are attributed to core 1 and no samples were collected for classic on any other core.



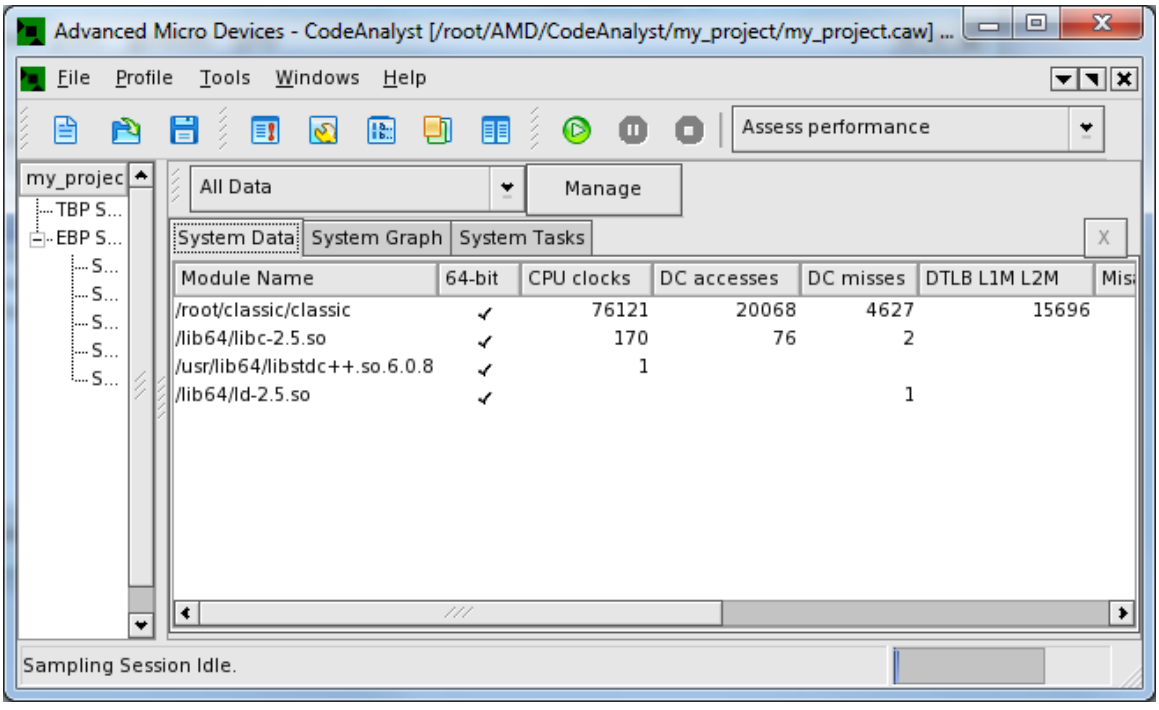
2.7.7. Process Filter

When the checkbox **Apply Process Filter** is selected, CodeAnalyst will filter out all other processes except the ones being specified in Launch. Users can also specify a list of processes to be included in post-processing using the Advance Filter.

Figure 2.43. Process Filter Dialog



Note that a profile is collected for all processes system-wide. However, only the profile specified in the Advance Filter is being processed. This can also help reduce the post-processing wait time in some cases. The following figure shows a profiling session with process filter enabled. Here, only `/root/classic` and `/lib64/libc-2.4.so` are shown since application "classic" depends on the standard "libc" library.



2.8. CodeAnalyst and OProfile

CodeAnalyst leverages a third-party profiling tool called OProfile [http://oprofile.sourceforge.net/news/]. CodeAnalyst provides graphical user inteface which communicates with Oprofile kernel module and Oprofile daemon in order to collect profile data. OProfile also provides command-line utilities for CodeAnalyst. CodeAnalyst is designed to work with original OProfile which is often available on several Linux distributions. CodeAnalyst also provide a modified OProfile kernel module, daemon, and utilities which include additional features and supports for latest AMD processors.

Opcontrol is a command-line tool from OProfile used to control a profiling session. It allows users to configure, start, and stop data collection. Users can invoke **opcontrol** from within a script, which is useful when the task requires multiple profile runs. To see the list of command line options, simply run **opcontrol --help** at command prompt. To identify the CodeAnalyst provided version of "opcontrol" tool, simply run **opcontrol --version**.

The profiling results from **opcontrol** can be imported into the CodeAnalyst Graphical User Interface (GUI) for analysis. A CodeAnalyst project must be created to import and view performance data that was collected using the command line utility. Please see Section 2.5, “Importing Profile Data into CodeAnalyst” for more information.

2.8.1. Profiling with OProfile Command-Line Utilities

The command line switches to **opcontrol** set up and control performance data collection. Analysis often concentrates on the performance of a particular program. The ability to control profiling from a command file offers considerable flexibility when writing test scripts. Specific test cases can be encapsulated into individual script files. The following is an example of how to use opcontrol in a script:

Figure 2.44. Command line switches to opcontrol

```
#!/bin/bash

# Reset Oprofile
opcontrol --reset

# Specify no vmlinux
opcontrol --no-vmlinux

# Specify oprofile merge option
opcontrol --separate=lib,kernel,cpu

#Specify event 0x76 0xC0 0x40 0x41
opcontrol --event=CPU_CLK_UNHALTED:100000:0:1:1  \
        --event=RETIRED_INSTRUCTIONS:100000:0:1:1  \
        --event=DATA_CACHE_ACCESSES:10000000:0:1:1 \
        --event=DATA_CACHE_MISSES:100000:0:1:1

# Start profiling
opcontrol --start

# Run target application or scripts
/root/testScript.sh

# Stop profiling
opcontrol --shutdown
```

2.8.1.1. Time-Based Profiling

Time-based profiling works by collecting samples at specified time intervals. Over a period of time, the samples collected can show which blocks of code use the most processor time. See Section 3.2, “Time-Based Profiling Analysis” for further information.

The timer interval determines how often a TBP sample is taken . On Linux, Time-based profiling uses event **CPU_CLK_UNHALTED** (performance counter event 0x76) which represents the amount of running time of a processor i.e. CPU is not in a halted state. This event allows system idle time to be automatically factored out from IPC (or CPI) measurements, providing the OS halts the CPU when going idle. The time representation (in seconds or millisecond) can be calculated from the processor clock speed. For instance, on a processor running at clock speed 800MHz, to specify 1 millisecond time interval of time-based profiling using the opcontrol tool, we input

opcontrol --event=CPU_CLK_UNHALTED:800000::1:1

2.8.1.2. Event-Based Profiling

Event-based profiling works by using performance counters in the processor to count the number of times a specific processor event occurs. When the specified counter threshold of an event is reached, Oprofile collects a sample from the processor. Up to four events can be profiled in a given session,and

each event can be assigned a different counter threshold. EBP requires an APIC-enabled system. See Section 3.3, “Event-Based Profiling Analysis” for further information.

To successfully use EBP, the user needs to consult the performance monitor event tables. See the section on Section 9.1, “Performance Monitoring Events (PME)” or the **"BIOS and Kernel Developer's Guide (BKDG)"** for the AMD processor in your test platform. For a general description of how to use these performance monitoring features, refer to the AMD64 Architecture Programmer's Manual, Volume 2, order# 24593, "Debug and Performance Resources" section.

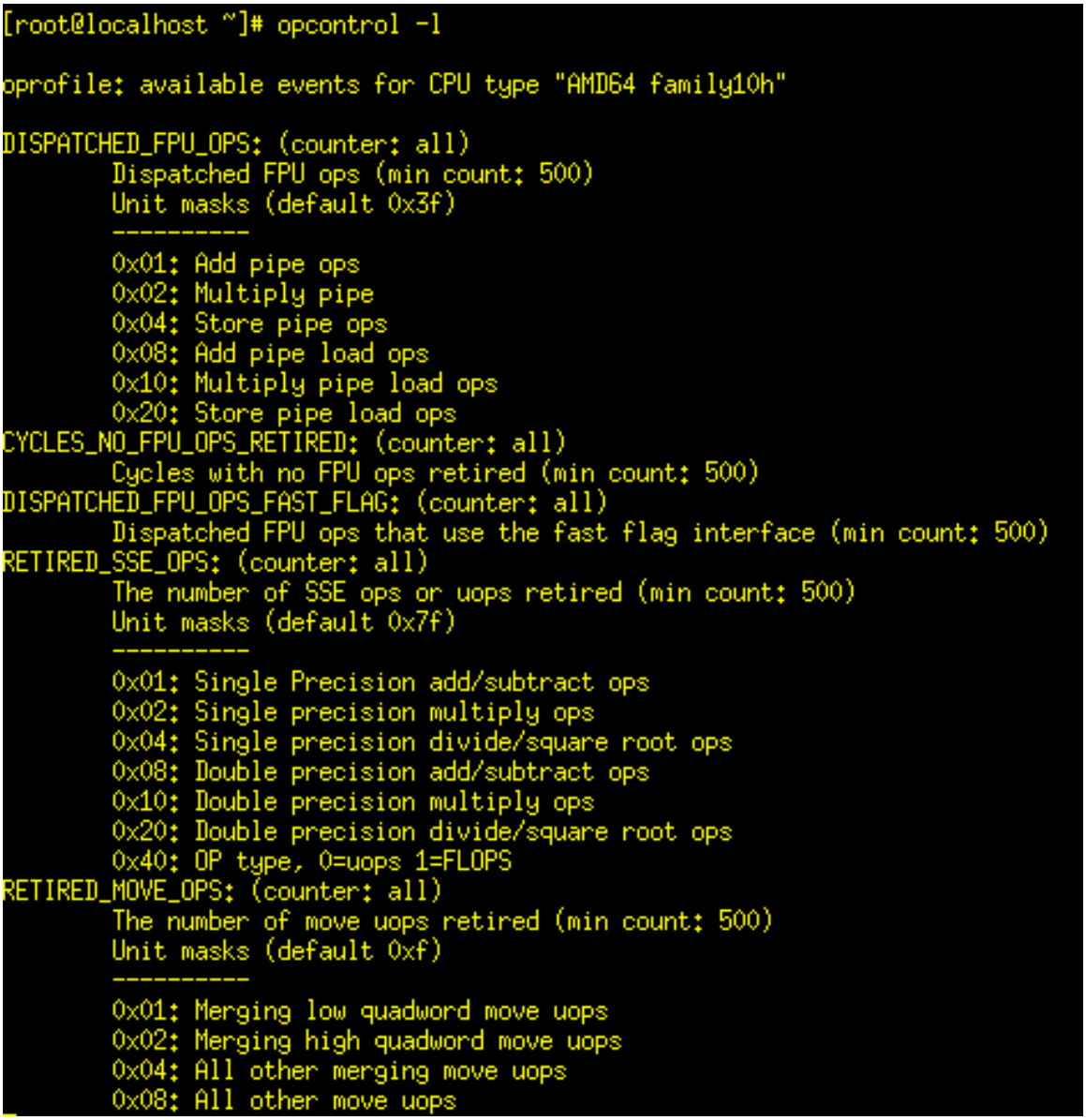
The Event Select, Unit Mask and Event Count (sampling period) must be specified for each event to be measured. The Oprofile utility accepts event specifications that are formatted in the following manner:

[OPROFILE_EVENT_NAME]:[Count]:[Unit mask]:[Kernel]:[User]

- **[OPROFILE_EVENT_NAME]** specifies the name of event to be profiled.
- **[Unit Mask]** is a two digit, hexadecimal value which specifies the Unit Mask value for the event.
- **[Count]** is a decimal number that specifies the Event Count (sampling period.)
- **[Kernel]** 0 or 1 to specify kernel-space profiling.
- **[User]** 0 or 1 to specify user-space profiling.

A complete list of events can be viewed using command **opcontrol -l**.

Figure 2.45. Listing events: opcontrol -l



Consider, for example, the DCache Refill From L2 or System event which can be used to measure only refills from system memory through the use of a Unit Mask that qualifies the event. The Event name is "DATA_CACHE_REFILLS_FROM_L2_OR_NORTHBRIDGE" and a Unit Mask value of 0x01 measures only refills from system memory. Using an Event Count of 25,000, the full opcontrol event specification is:

**opcontrol **

-e DATA_CACHE_REFILLS_FROM_L2_OR_NORTHBRIDGE:25000:0x1:1:1

The Retired Instructions event (Event Select 0x0C0) does not require a Unit Mask. Using an Event Count of 250,000, the full opcontrol event specification is:

opcontrol --event=RETIRED_INSTRUCTIONS:250000::1:1

The Event Count field of the event specification determines how many times the event is to be counted before an EBP sample is collected. This field is important because some events happen so frequently that the processing of the samples becomes very slow or collecting the samples can cause the system to stop responding. This latter scenario may occur if the specified Event Count is too small for an

event. For example, event 0x4000 (DCache Access) happens quite frequently, so specifying a count of 10000 will generate a significant number of samples during a normal 5-second profiling session. It is advisable to specify a large count number when profiling an event for the first time, and allow for later adjustments of the count to get more statistically useful data.

The following example command measures the Data Cache Accesses event (Event Select 0x040) using an Event Count of 100,000:

**opcontrol **

--event=DATA_CACHE_ACCESSES:100000::1:1

The user is allowed to specify up to four events on the command line. For example, to specify events DATA_CACHE_ACCESSES and RETIRED_INSTRUCTIONS in the same session with reasonable counts, enter the following command:

**opcontrol **

**--event=DATA_CACHE_ACCESSES:100000::1:1 **

--event=RETIRED_INSTRUCTIONS:25000::1:1

After profiling has begun, oprofile prints out the confirmation messages and returns to command prompt. To stop, the profiling session, simply run opcontrol with the corresponded option flags as listed in **opcontrol --help**.

2.8.1.3. Instruction-Based Sampling

Instruction-Based Sampling collects performance data on instruction fetch (IBS fetch sampling) and macro-op execution (IBS op sampling.) IBS fetch sampling provides information about instruction cache (IC), instruction translation lookaside buffer (ITLB) behavior, and other aspects of the process of fetching instructions. IBS op sampling provides information about the execution of macro-ops that are issued from AMD64 instructions. IBS op data is wide-ranging and covers branch and memory access operations. See Section 3.4, “Instruction-Based Sampling Analysis” for more information.

IBS fetch sampling and IBS op sampling are controlled by the specifying

opcontrol --event=<IBS-Fetch-event-name>:<Fetch-count>:<Fetch-option-mask>:1:1

or

opcontrol --event=<IBS-Op-event-name>:<Op-count>:<Op-option-mask>:1:1

The count takes a single decimal value which is the fetch/op interval (sampling period) for IBS fetch/op sampling. The IBS fetch sampling counts completed fetches to determine the next fetch operation to monitor and sample. The IBS op sampling counts processor cycles to determine the next macro-op to monitor and sample. IBS fetch and op sampling may be enabled independently, or at the same time. For instance,

**opcontrol **

**--event=IBS_FETCH_ALL:250000:0:1:1 **

--event=IBS_OP_ALL:250000:0:1:1

Please see **opcontrol --list** for list of available IBS-derived events and option mask. Also see Section 9.3, “Instruction-Based Sampling Derived Events” for description of IBS-derived events.

NOTE: All IBS-Fetch events must have same counts and option-mask. The rule also applies for IBS-op events.

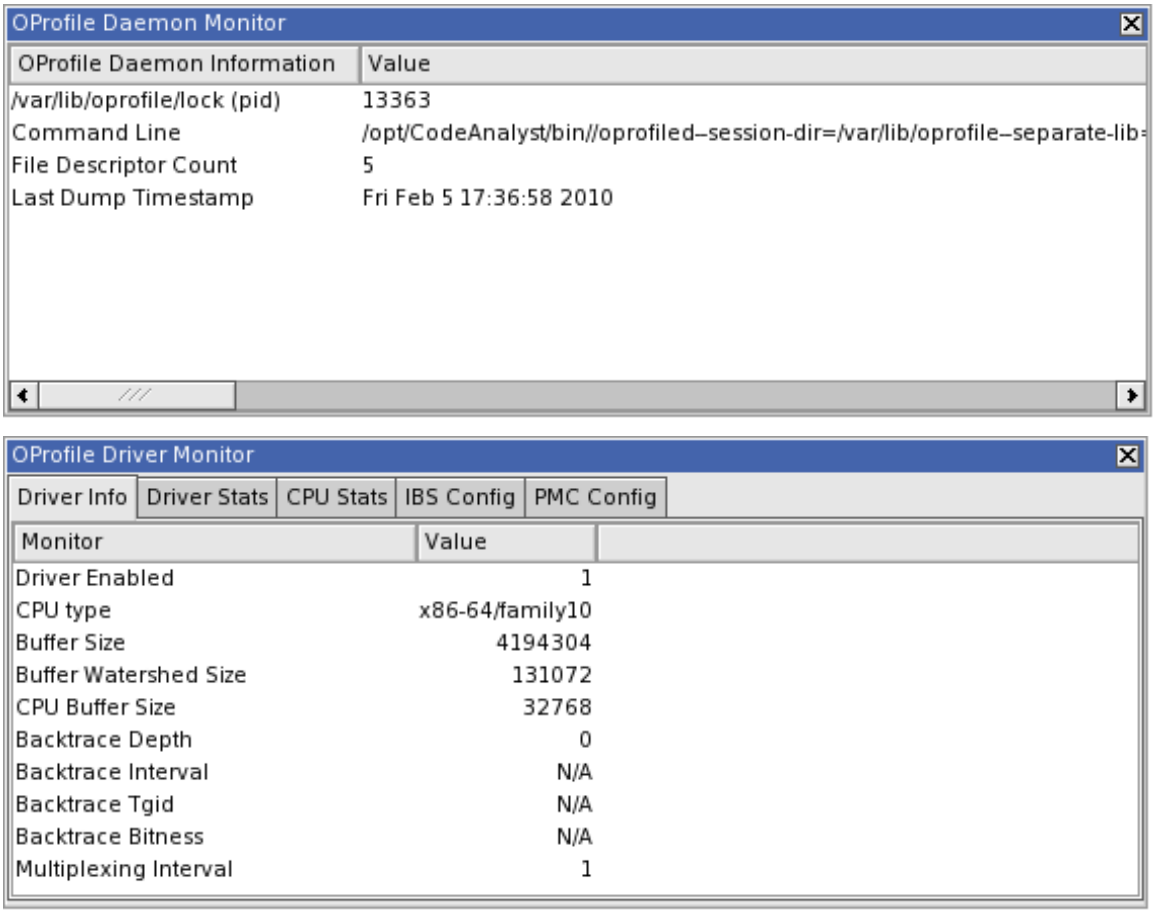
2.8.2. Importing and Viewing Profile Data

Profiles taken by using the command line utility can be imported into a CodeAnalyst project. With default configuration, opcontrol creates profiles in `/var/lib/oprofile/samples/current` directory. It contains the samples collected by Oprofile daemon.

The profile data can be imported into the CodeAnalyst GUI for further review and interpretation. Profile data are imported into a CodeAnalyst project. See Section 8.3, “Tutorial - Creating a CodeAnalyst Project” in order to create a project. The section Section 2.5, “Importing Profile Data into CodeAnalyst” illustrates the process of importing profile data.

2.8.3. OProfile Daemon/Driver Monitoring Tool

Figure 2.46. OProfile Daemon/Driver Monitoring Tool



CodeAnalsyt provides a utility to monitor OProfile daemon and driver activities. The tool accesses information in `/dev/oprofile/` and `/var/lib/oprofile/` directories, then present statistics in realtime. This tool helps provide insight into data collection subsystem. The tool is a dock window which can be moved around the main window or pulled out as a stand-alone dialog.

Chapter 3. Types of Analysis

3.1. Types of Analysis

AMD CodeAnalyst provides several types of analysis. Each kind of analysis provides information about certain aspects of program behavior and performance. The following sections provide details about the kinds of analysis provided by AMD CodeAnalyst:

- Section 3.2, “Time-Based Profiling Analysis”
- Section 3.3, “Event-Based Profiling Analysis”
- Section 3.4, “Instruction-Based Sampling Analysis”
- Section 3.5, “Basic Block Analysis”
- Section 3.6, “In-Line Analysis”
- Section 3.7, “Session Diff Analysis”

Time-based, Event-based, and Instruction-based sampling(IBS) analysis require different types of profile data collecting at run-time. The section on time-based profiling has tips that apply to the other forms of analysis that use statistical sampling (EBP and IBS).

The basic-block and in-line analysis are static analysis which inspects binary modules and allows users to better analyze the compiler generated assembly code by identifying basic-block and inline instance.

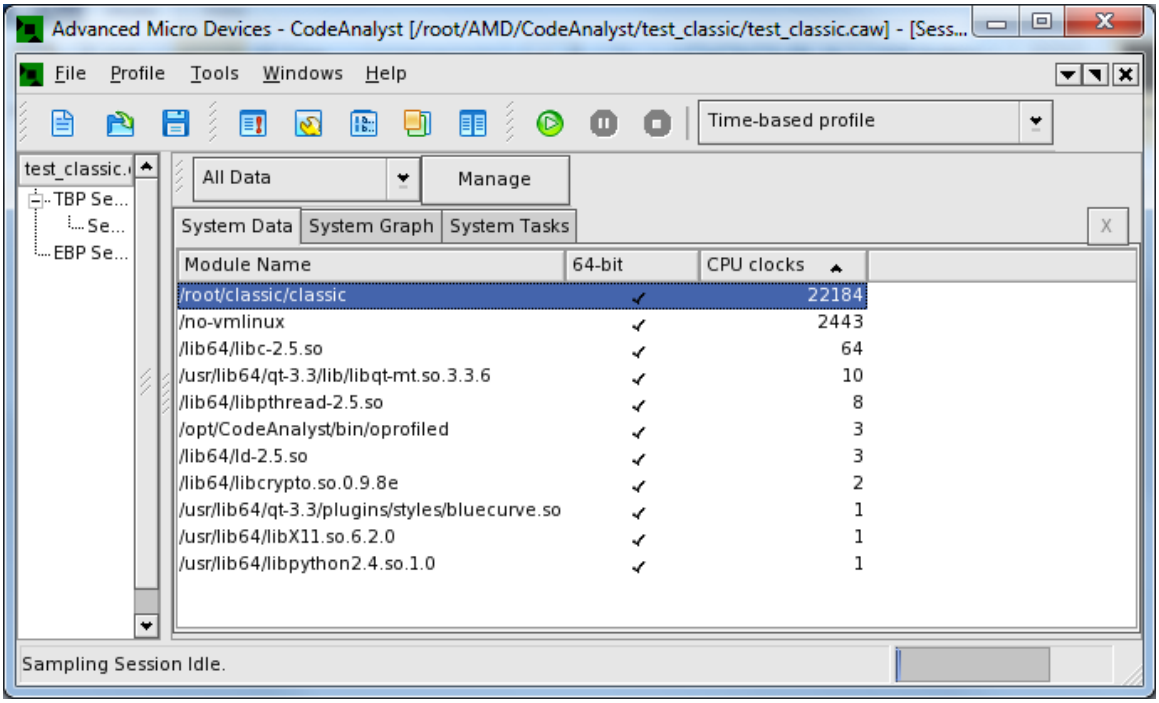
The Session Diff Analysis allow users to visualy compare any two profiling session at modules, functions, and assembly level.

3.2. Time-Based Profiling Analysis

Time-based profiling (TBP) identifies the hot-spots in a program that are consuming the most time. Hot-spots are good candidates for further investigation and optimization.

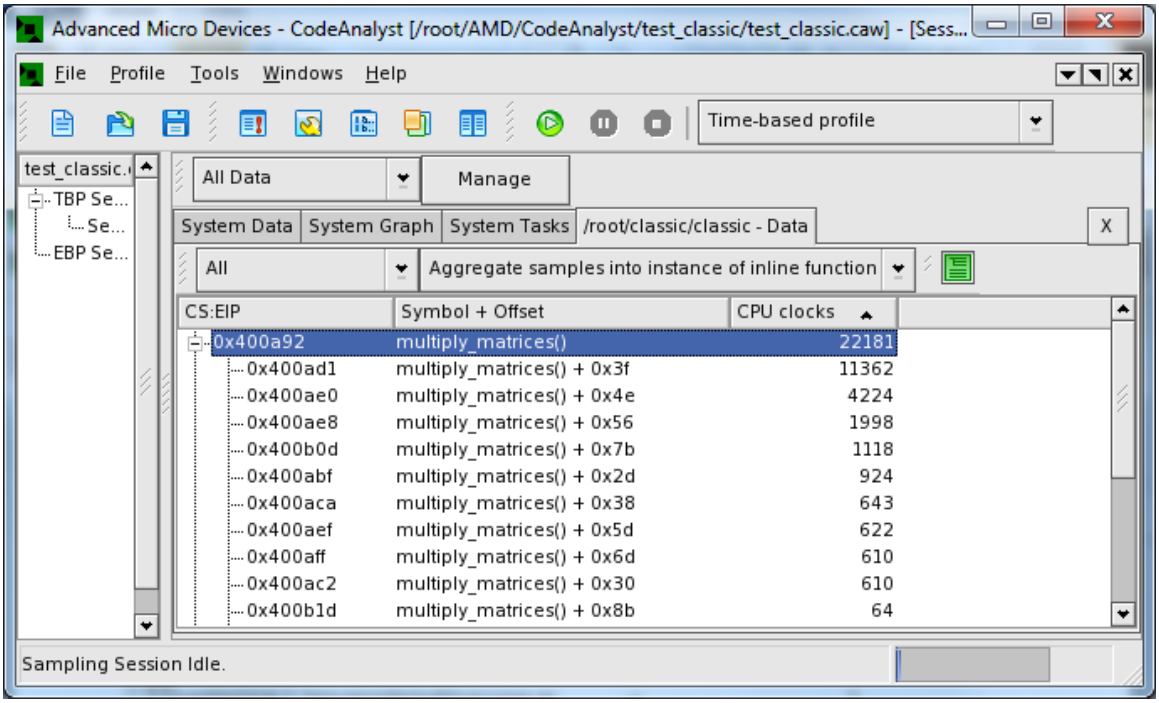
Time-based profiling is system-wide, so a hot-spot can be identified anywhere in the system, not just an application under test. Any software component (an executable image, dynamically loaded library, device driver, or even the operating system kernel) that executes during the measurement period may be sampled. System-wide data collection conveniently handles applications consisting of parallel processes without any special configuration.

Figure 3.1. Time spent in each software module (TBP)



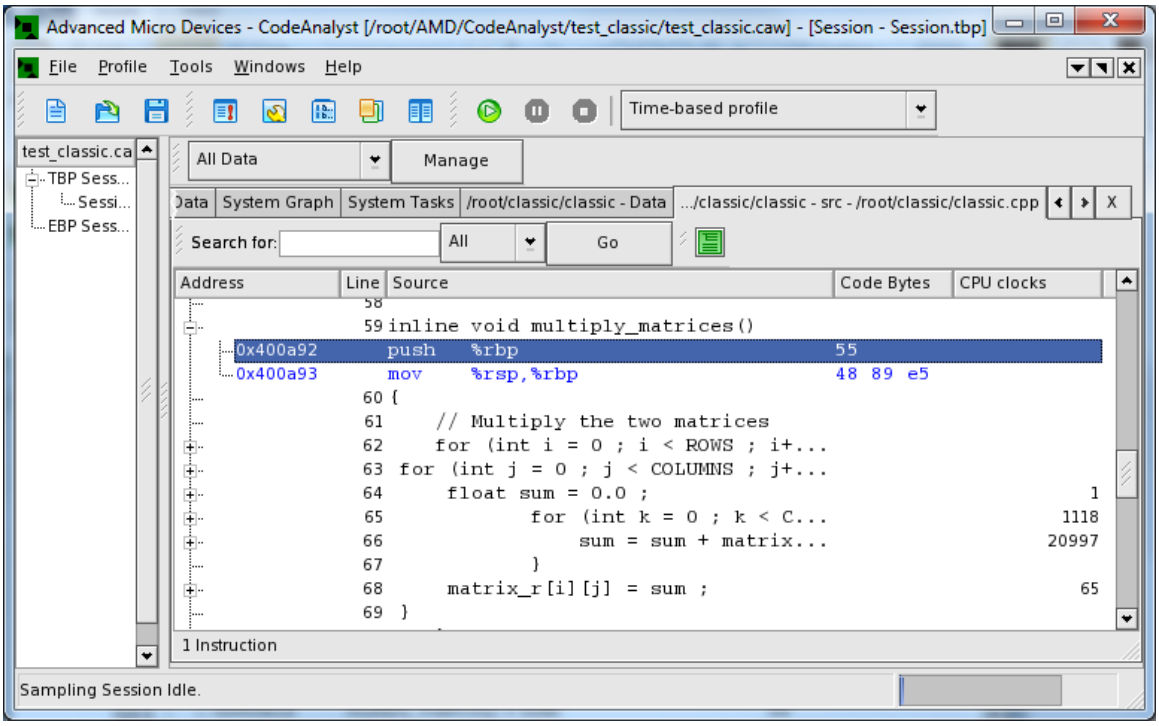
CodeAnalyst reports performance results in one or more tabs in the workspace. Results are broken out and summarized by module, process, function, source line, or instruction.

Figure 3.2. Time spent in each function within an application (TBP)



CodeAnalyst supports drill-down to source lines and instructions. Source-level presentation of performance results requires symbolic (debug) information. See Section 1.1.1, “Preparing an Application for Profiling”.

Figure 3.3. Time spent at source-level hot-spot (TBP)



3.2.1. How Time-Based Profiling Works

Time-based profiling uses statistical sampling to collect and build a program profile. CodeAnalyst handles all of the details and mechanics of collecting and building a profile. However, simple knowledge of the TBP sampling process is helpful.

When time-based profiling is enabled and started, CodeAnalyst configures a timer that periodically interrupts the program executing on a processor core. When a timer interrupt occurs, a sample is created and saved for post-processing by the CodeAnalyst GUI. Post-processing builds up a kind of histogram, which is a profile of what the system and its software components were doing. The most time-consuming parts of a program will have the most samples because, most likely, the program is executing in those regions when a timer interrupt is generated and a sample is taken.

On CodeAnalyst Linux, Time-based profiling uses event **CPU_CLK_UNHALTED** (performance counter event 0x76) which represents the amount of running time of a processor i.e. CPU is not in a halted state. This event allows system idle time to be automatically factored out from IPC (or CPI) measurements, providing the OS halts the CPU when going idle. The time representation (in seconds or millisecond) can be calculated from the processor clock speed. For instance, on a processor running at clock speed 800MHz, to specify 1 millisecond time interval of time-based profiling.

3.2.2. Sampling Period and Measurement Period

Because TBP relies upon statistical sampling, collecting enough samples to reason about program behavior is important. The number of TBP samples collected during an experimental run depends upon:

- How frequently samples are taken (the sampling period), and
- The length of the time during which samples are taken (the measurement period.)

The frequency of sample taking is controlled by the timer **interval**. This quantity is sometimes called the "sampling period." The default timer interval is 1 millisecond, meaning that a TBP sample is taken

on a processor core approximately every one millisecond of wall clock-time. The timer interval can be changed by editing the **Current time-based profile** profile configuration. With the default timer interval, roughly 1,000 TBP samples are taken per second for each processor core.

With a shortened interval time, samples are taken more often and more samples are taken within a given, fixed-length measurement period. However, the amount of effort expended to take samples (known as **overhead**) will increase, placing the test system under a higher load. The process of taking samples and the overhead have an intrusive effect that perturbs the test workload and may bias statistical results.

As to the second factor—the length of time during which samples are taken—samples accumulate for as long as data is collected. The measurement period depends upon the overall execution time of the workload and the way in which CodeAnalyst data collection is configured. Using either the **Session Settings** or command line utility options, CodeAnalyst can be configured to collect samples for all or part of the time that the test workload executes. If program run-time is short (less than 15 seconds), it may be necessary to increase program run-time by using a larger data set or more loop iterations to obtain a statistically useful result. Extending the duration of the measurement period by changing the Session Settings or options to the CodeAnalyst command line utility may need to be done.

Deciding how many samples are enough requires judgment and a working knowledge about the characteristics of the workload under test. Scientific applications often have tight inner loops that are executed several times. In these situations, samples accumulate rapidly within the inner loops and even a fairly short run-time yields a statistically useful number of samples. Other workloads, like transaction processing, have few intense inner loops and the profiles are relatively "flat." For flat workloads, a longer measurement period is required to build up samples in code regions of interest.

3.2.3. Predefined Profile Configurations

CodeAnalyst provides predefined **profile configurations** to make configuration of time-based profiling and other kinds of analysis convenient. The predefine configuration for time-based profiling is called "Time-based profile." CodeAnalyst also provides a configuration named "Current time-based profile" where the timer interval (sampling period) can be changed.

3.3. Event-Based Profiling Analysis

Event-based profiling (EBP) helps identify the root cause for CPU- and memory-related performance issues. EBP uses the performance monitoring hardware in AMD processors to count the number of occurrences of hardware events. The kind and frequency of these events may indicate the presence of a pipeline bottleneck, poor memory access pattern, poorly predicted conditional branches, or some other performance issue. Once hot-spots are found through time-based profiling, EBP is used to follow-up and investigate the hot-spots in order to identify and exploit opportunities for optimization.

Figure 3.4. Retired instructions, DC accesses and misses per software module (EBP)

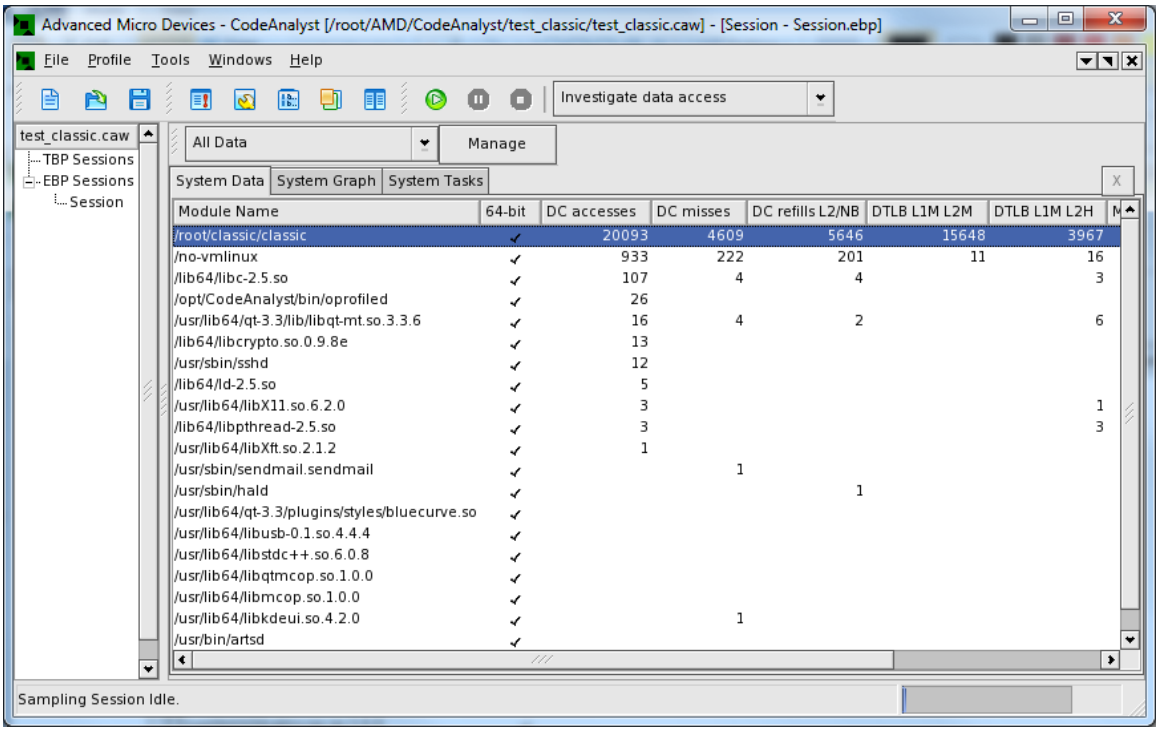
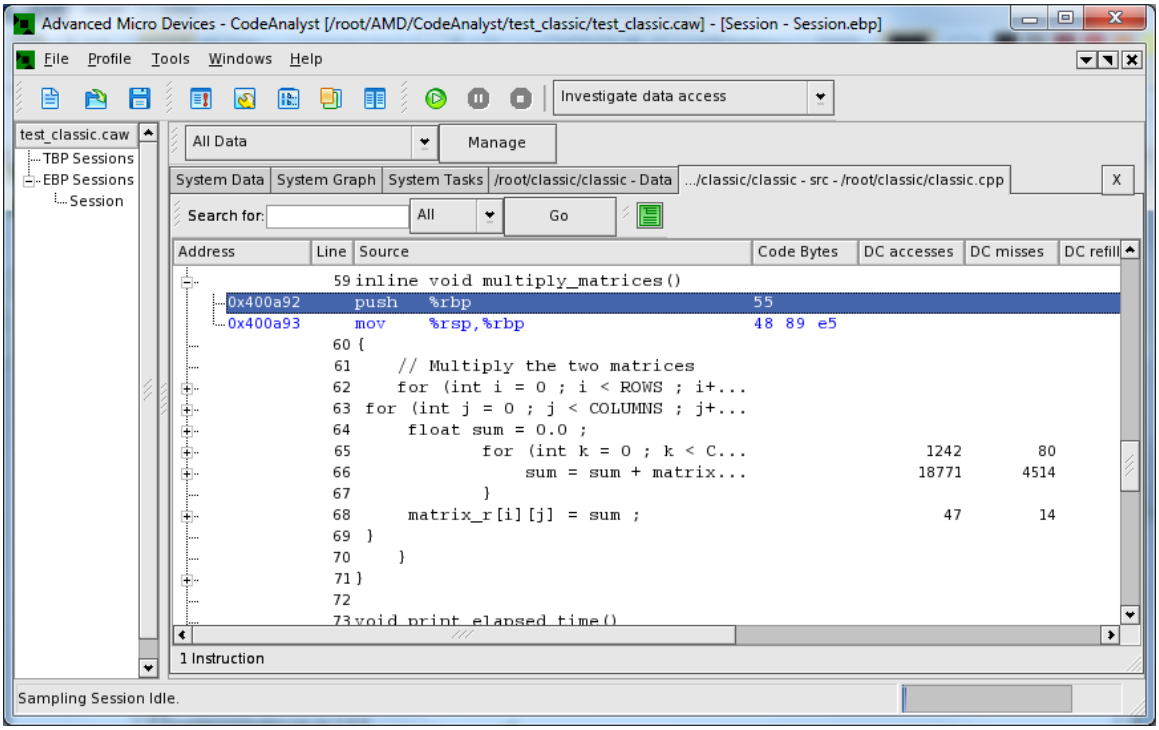


Figure 3.5. Retired instructions, DC accesses and misses for source-level hot-spot (EBP)



AMD processors are equipped with performance monitoring counters (PMC). Each counter may count exactly one hardware event at a time. A hardware event is a condition (or change in hardware condition) like CPU clocks, retired x86 instructions, data cache accesses, or data cache misses. The number of counters and the hardware events that can be measured are processor-dependent. The CodeAnalyst online help provides a quick guide to the events that are available for each AMD processor family. See

"Performance Monitoring Events (PME)" for descriptions of the events supported by AMD processors. However, you should consult the **BIOS and Kernel Developer's Guide** for the AMD processor in your test platform for the latest information. The number of events and, in some cases, the event behavior may vary by revision within a processor family as well.

3.3.1. How Event-Based Profiling Works

Like time-based profiling, event-based profiling relies upon statistical sampling to build a program profile. CodeAnalyst handles the details of PMC configuration and sampling. However, the following short description of how CodeAnalyst performs event-based profiling may help to understand how to use CodeAnalyst more effectively. Each counter must be configured with:

- An event to be measured (specified by event select and unit mask values,
- An event count (sampling period),
- Choice of OS-space sampling, user-space sampling, or both, and
- Choice of edge- or level-detect.

Some AMD processor families support additional configuration parameters and CodeAnalyst offers these parameters when they are supported on the test platform. Once a PMC has been configured and sampling is started, the counter counts the event to be measured until the event count (sampling period) is reached. At that time, an interrupt generates and an EBP sample taken. The EBP sample contains a timestamp, the kind of event that triggered the interrupt, the identifier of the process that was interrupted, and the instruction pointer where the program will restart after the return from the sampling interrupt. CodeAnalyst uses this data along with information from the executable images, debug information, and source code to accumulate and display a profile for each executing software component (process, module, function, source line, or instruction.)

3.3.2. Sampling Period and Measurement Period

Since EBP is a statistical method, it also depends upon a statistically significant quantity of samples in order to support reasoning about program behavior. As discussed in Section 3.2, "Time-Based Profiling Analysis", the number of samples collected depends upon the sampling period (the event count parameter) and the measurement period (the length of time during which samples are collected). The number of samples collected can be increased by using a smaller sampling period or by increasing the length of time that samples are taken.

Use of a smaller sampling period increases data collection overhead. Since data collection must be performed on the same platform as the test workload, more frequent sampling increases the intrusiveness of event-based profiling and the sampling process adversely affects shared hardware resources like instruction and data caches, translation lookaside buffers and branch history tables. Extremely small sampling periods may also cause system instability. Start off conservatively and slowly decrease the sampling period for an event until the appropriate volume of samples is generated.

An additional complicating factor when choosing the sampling period for an event is the behavior of the workload itself. Some workloads are CPU-intensive while other workloads are memory-intensive. Some workloads may be CPU-intensive and require high memory bandwidth to **stream** data into the CPU. For example, a CPU-intensive application that performs few memory access operations will cause relatively few data-cache miss events simply because it does not access memory very often. The characteristics of the workload may even vary by **phase** where the phase setting up a computation has a different behavior from the computation phase itself. Thus, the workload behavior determines the frequency of certain kinds of events and changes to the sample period may be necessary in practice.

3.3.3. Event Multiplexing

As mentioned earlier, the number of available performance counters is processor-dependent. AMD Family 10h processors, for example, provide four performance counters. The number of available

performance counters determines the number of events that can be measured together at one time. Ordinarily, this would limit the number of events that can be measured in a single experimental run. However, CodeAnalyst for Linux®, allows users to set more than four performance counters within a profiling session. See Section 2.4, “Event Counter Multiplexing” for further details.

NOTE: The semantic for Event Multiplexing used in CodeAnalyst for Linux is different from the one in Windows.

3.3.4. Predefined Profile Configurations

To make the process of configuration easier, CodeAnalyst provides several predefined **profile configurations** in which the choice of events and sampling periods have already been made. These predefined profile configurations are:

- Assess performance
- Investigate data access
- Investigate instruction access
- Investigate L2 cache access
- Investigate branching

CodeAnalyst also provides a configuration named "Current event-based profile" that allows choosing your own events to measure, change sampling periods (event counts), and change other EBP configuration parameters. See Section 4.4, “Predefined Profile Configurations” for more information.

3.4. Instruction-Based Sampling Analysis

Instruction-Based Sampling (IBS) uses a hardware sampling technique to generate event information which is similar to that produced by Section 3.3, “Event-Based Profiling Analysis” Instruction-Based Sampling can be used to identify and diagnose performance issues in program hot-spots. IBS has these advantages:

- Events are precisely attributed to the instructions that caused the events.
- IBS produces a wealth of event data in a single test run.
- Latency is measured for key performance factors such as data cache miss latency.

The information provided through Instruction-Based Sampling covers the most common kinds of information needed for program performance analysis. Event-based profiling, however, offers a wider range of events that can be monitored, such as events related to HyperTransport™ links.

The processor pipeline stages can be categorized into two main phases—instruction fetch and execution. Each instruction fetch operation produces a block of instruction data that is passed to the decoder stages in the pipeline. The decoder identifies AMD64 instructions in the fetch block. These AMD64 instructions are translated to one or more macro-operations, called "macro-ops" or "ops," that are executed in the execution phase.

Figure 3.6. IBS op samples for each software module

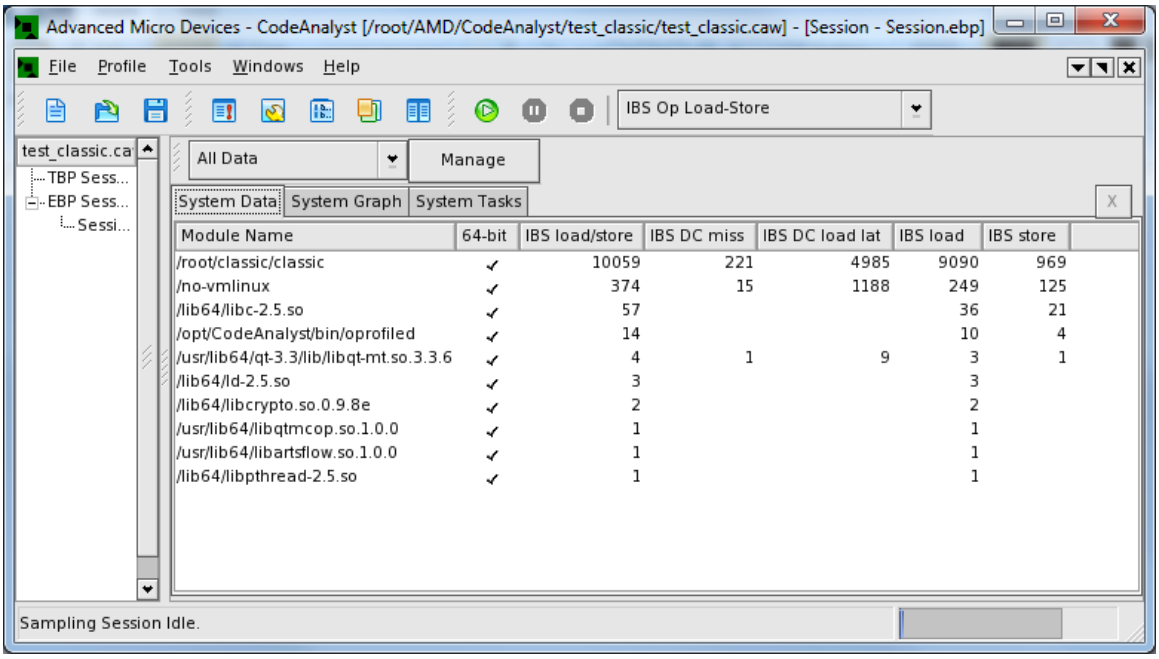
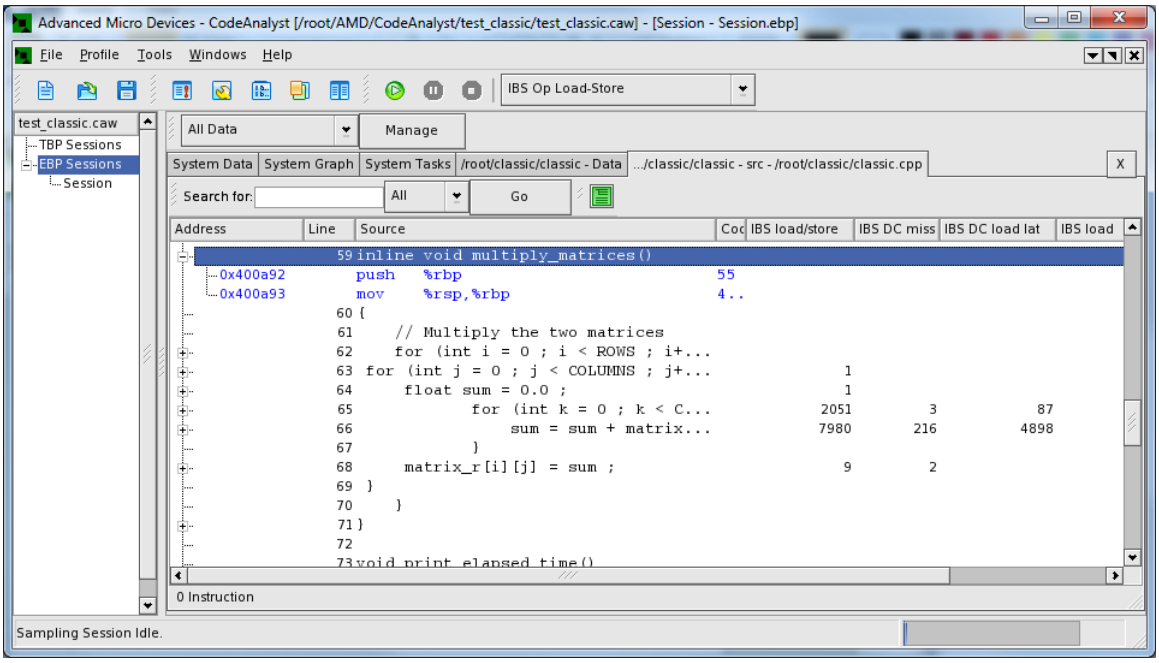


Figure 3.7. Attribution of IBS op samples to source-level hot-spot



Instruction-Based Sampling provides separate means to sample fetch operations and macro-ops since the fetch and execution phases of the pipeline treat these entities separately. IBS fetch sampling and IBS op sampling may be enabled and collected separately, or both may be enabled and collected together.

3.4.1. IBS Fetch Sampling

IBS fetch sampling is a statistical sampling method. IBS fetch sampling counts completed fetch operations. When the number of completed fetch operations reaches the maximum fetch count (the sampling period), IBS tags the fetch operation and monitors that operation until it either completes or aborts.

When a tagged fetch completes or aborts, a sampling interrupt is generated and an IBS fetch sample is taken. An IBS fetch sample contains a timestamp, the identifier of the interrupted process, the virtual fetch address, and several event flags and values that described what happened during the fetch operation. Like time-based profiling and event-based profiling, CodeAnalyst uses the IBS sample data and information from the executable images, debug information, and source to build an profile IBS for software components executing on the system. (Instruction-Based Sampling is also system-wide.)

The event data reported in an IBS sample includes:

- Whether the fetch completed or aborted,
- Whether address translation initially missed in the level one (L1) or level two (L2) instruction translation lookaside buffer (ITLB),
- The page size of the L1 ITLB address translation (4K, 2M),
- Whether the fetch initially missed in the instruction cache (IC), and
- The fetch latency (number of processor cycles from when the fetch was initiated to when the fetch completed or aborted).

Event-based profiling would require several different counters in order to collect as much information as IBS. Further, the fetch address precisely identifies the fetch operation associated with the hardware events.

The IBS fetch address may be the address of a fetch block, the target of a branch, or the address of an instruction that is the fall-through of a conditional branch. A fetch block does not always start with a complete valid AMD64 instruction. This situation occurs when an AMD64 instruction straddles two fetch blocks. In this case, CodeAnalyst associates the IBS fetch sample with the AMD64 instruction in the first (preceding) fetch block.

The terms "killed," "attempted," "completed," and "aborted" refer to specific hardware conditions. A fetch operation may be abandoned before it delivers data to the decoder. A fetch may be abandoned due to a control flow redirection, and it may be abandoned at any time during the fetch process. A fetch abandoned before initial access to the ITLB (before address translation) is not regarded as useful for analysis. These early abandoned fetches are called **killed** fetches. CodeAnalyst filters out killed fetches. The fetch operations remaining after killed fetches are eliminated are called **attempted** fetches since these fetches represent valid attempts to obtain instruction data.

A **completed** fetch is an attempted fetch that successfully delivered instruction data to the decoder. An **aborted** fetch is an attempted fetch that did not complete.

Note: Instruction fetch is an aggressive, speculative activity and even instruction data produced by a completed fetch may not be used.

3.4.2. IBS Op Sampling

IBS op sampling selects, tags, and monitors macro-ops as issued from AMD64 instructions. Two options are available for selecting ops for sampling.

- **Cycles-based** selection counts CPU clock cycles. The op is tagged and monitored when the count reaches a threshold (the sampling period) and a valid op is available.
- **Dispatched op-based** selection counts dispatched macro-ops. When the count reaches a threshold, the next valid op is tagged and monitored.
- In both cases, an IBS sample is generated only if the tagged op retires. Thus, IBS op event information does not measure speculative execution activity.

The execution stages of the pipeline monitor the tagged macro-op. When the tagged macro-op retires, a sampling interrupt is generated and an IBS op sample is taken. An IBS op sample contains a timestamp, the identifier of the interrupted process, the virtual address of the AMD64 instruction from which the op was issued, and several event flags and values that describe what happened when the macro-op executed. CodeAnalyst uses this and other information to build an IBS profile.

3.4.2.1. Bias in Cycles-Based IBS Op Sampling

A cycles-based selection generally produces more IBS samples than dispatched op-based selections. However, the statistical distribution of IBS op samples collected with a cycles-based selection may be affected by pipeline stalls and other time-dependent hardware behavior. The statistical bias is due to stalls at the decoding stage of the pipeline. If a macro-op is not available for tagging when the maximum op count is reached, the hardware skips the opportunity to tag a macro-op and starts counting again from a small, pseudo-random initial count.

From a practical perspective, the distribution of cycles-based IBS op samples may not be uniform across instructions with the same execution frequency (i.e., across instructions within the same basic block). The statistical distribution of IBS op samples collected with dispatched op-based selection is generally more uniform across instructions with the same execution frequency. This is a useful property in practice as IBS op statistics can be more readily used to make comparisons between instruction behavior. **The dispatched op-based selection is the preferred collection mode and should be used when available.**

Note: The cycles-based selection is supported on all IBS-capable AMD processors. The dispatched op-based selection is a newer IBS feature and is not supported on all IBS-capable AMD processors and is only available in AMD Family 10h processors, revision 4 and beyond. Refer to the relevant version of the AMD BIOS and kernel developer's guide for support details.

3.4.2.2. IBS Op Data

IBS op sampling reports a wide range of event data. The following values are reported for all ops:

- The virtual address of the parent AMD64 instruction from which the tagged op was issued,
- The tag-to-retire time (the number of processor cycles from when the op was tagged to when the op retired), and
- The completion-to-retire time (the number of processor cycles from when the op completed to when the op was retired.)

Attribution of event information is precise because the IBS hardware reports the address of the AMD64 instruction causing the events. For example, branch mispredicts are attributed exactly to the branch that mispredicted and cache misses are attributed exactly to the AMD64 instruction that caused the cache miss. IBS makes it easier to identify the instructions which are performance culprits.

Some ops implement branch semantics. Branches include unconditional and conditional branches, subroutine calls and subroutine returns. Event information reported for branch ops include:

- Whether the branch was mispredicted and
- Whether the branch was taken,

IBS also indicates whether a branch operation was a subroutine return and if the return was mispredicted. Some ops may perform a load (memory read), store (memory write), or a load and a store to the same memory address, as in the case of a read-op-write sequence. When an op performs a load and/or store, event information includes:

- Whether a load was performed,

- Whether a store was performed,
- Whether address translation initially missed in the L1 and/or L2 data translation lookaside buffer (DTLB),
- Whether the load or store initially missed in the data cache (DC),
- The virtual data address for the memory operation, and
- The DC miss latency when a load misses the DC.

Requests made through the Northbridge produce additional event information:

- Whether the access was local or remote and
- The data source that fulfilled the request.

A full list of IBS op event information appears in the section on IBS events. Hardware-level details can be found in the **BIOS and Kernel Developer's Guide** (BKDG) for the AMD processor in your test platform.

3.4.3. IBS-Derived Events

CodeAnalyst translates the IBS information produced by the hardware into **derived event** sample counts that resemble EBP sample counts. All IBS-derived events have "IBS" in the event name and abbreviation. Although IBS-derived events and sample counts look similar to EBP events and sample counts, the source and sampling basis for the IBS event information are quite different. Arithmetic should *never* be performed between IBS derived event sample counts and EBP event sample counts. It is not meaningful to directly compare the number of samples taken for events which represent the same hardware condition. For example, fewer IBS DC miss samples is not necessarily "better" than a larger quantity of EBP DC miss samples.

See Section 9.3, “Instruction-Based Sampling Derived Events” for descriptions of the IBS derived events.

3.4.4. Predefined Profile Configurations

AMD CodeAnalyst provides a predefined profile configuration called "Instruction-based sampling" which collects both IBS fetch and IBS op samples. It also provides the configuration named "Current Instruction-based profile " which can be changed and customized.

3.5. Basic Block Analysis

Basic block is a section of code that represents a serialized execution path that does not contain any type of control transfer instruction (i.e. jump or call). A basic block usually begins with the destination of a single or multiple control transfer instructions and ending with a control transfer instruction.

In the Module Data view, users can aggregate data by using basic block aggregation of samples. This mode of aggregation is enabled by selecting the **"Aggregate samples into basic blocks"** option in the **"CodeAnalyst Options"** dialog. CodeAnalyst examines each function to identify basic blocks and aggregates samples accordingly. Each basic block is denoted by a range of addresses using the format **"[StartAddr, StopAddr) : (Number of Load / Number of store)"**. The dialog also displays the number of load and store instructions within the basic block on the disassembly view tab.

Figure 3.8. CodeAnalyst Options

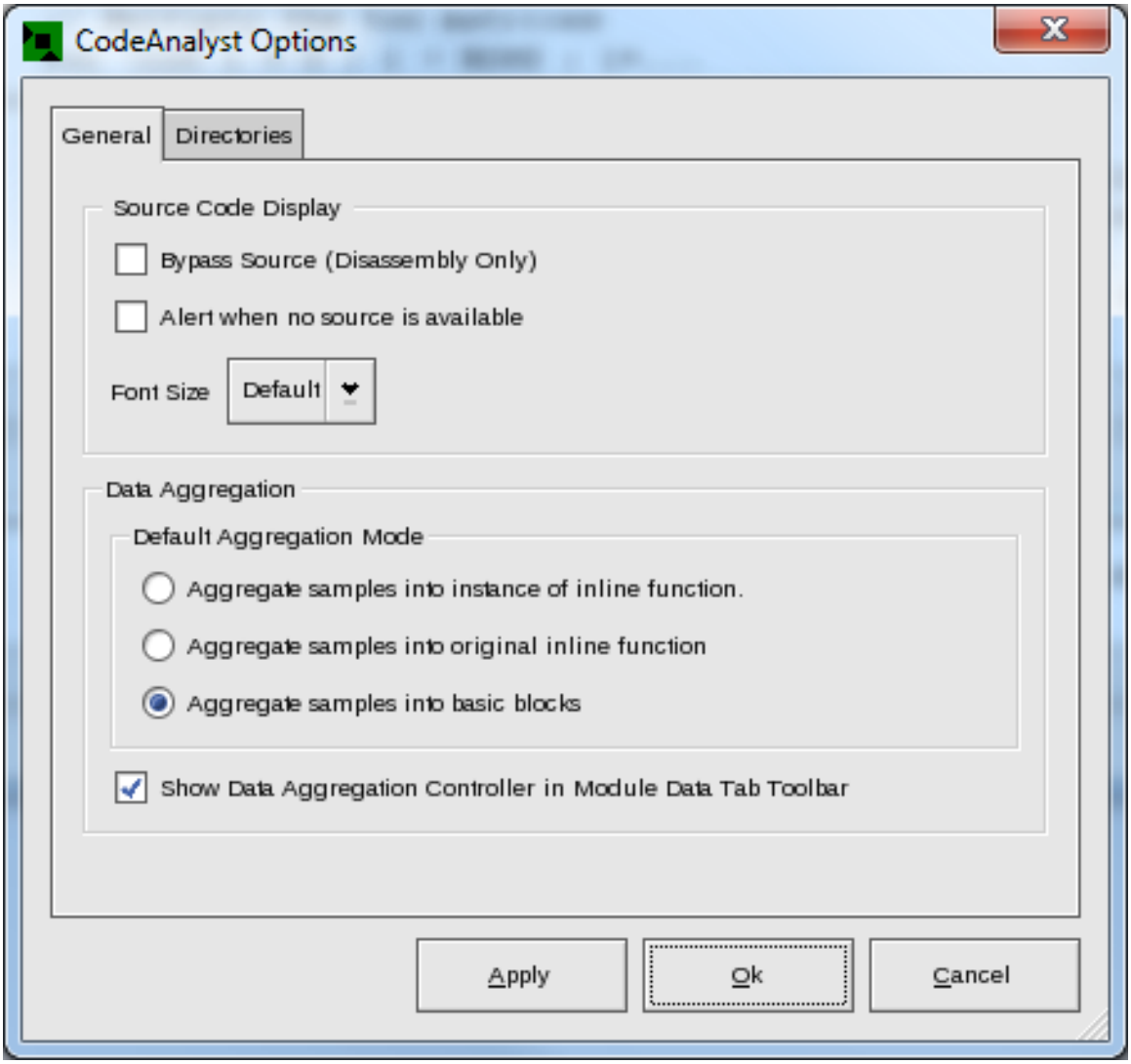
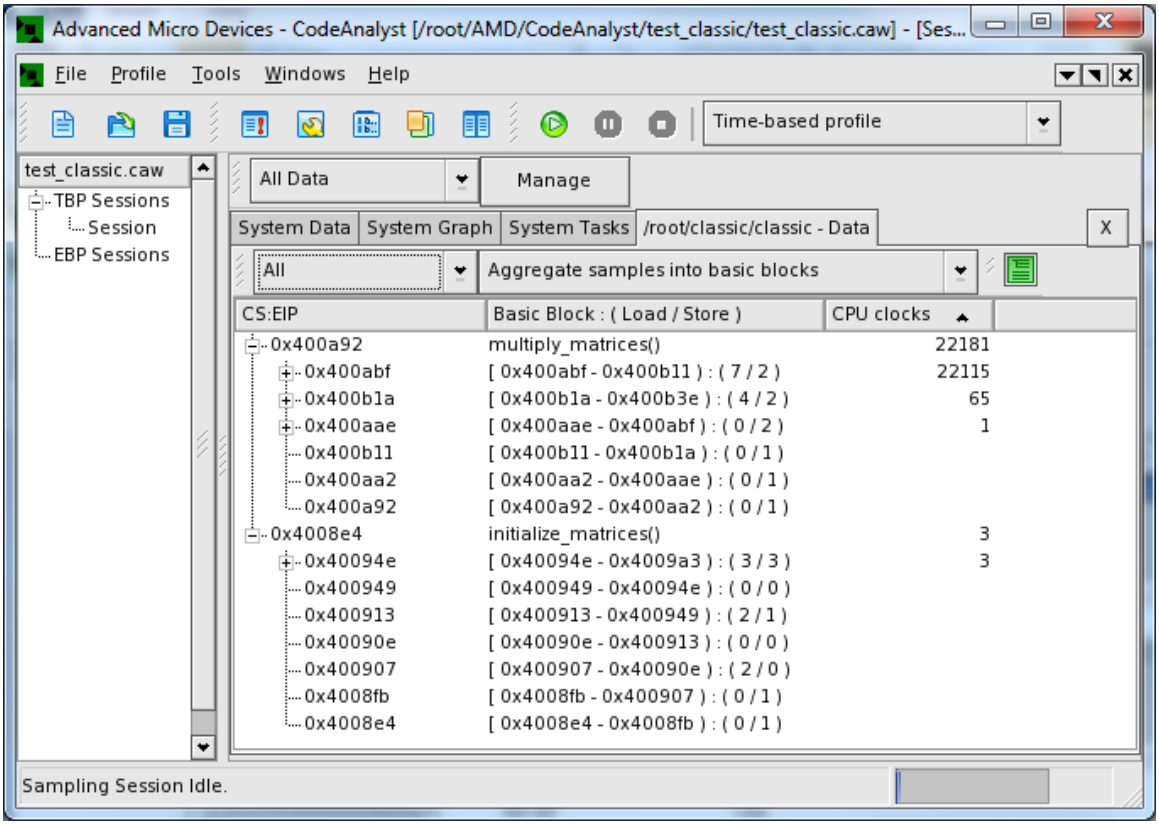


Figure 3.9. Module data view of basic block aggregation



In Disassembly View, each basic block is shown by interleaving different background colors of white and gray. Users can navigate through code execution path from one basic block to the previous or to the next basic block.

Right-click at the beginning of a basic block and the pop-up menu lists the source addresses that are usually the destination address of a control transfer instruction in some basic blocks.

Right-click at the end of a basic block and the pop-up menu lists the destination address of the control transfer instruction (see following image).

Figure 3.10. Basic block information in Disassembly View

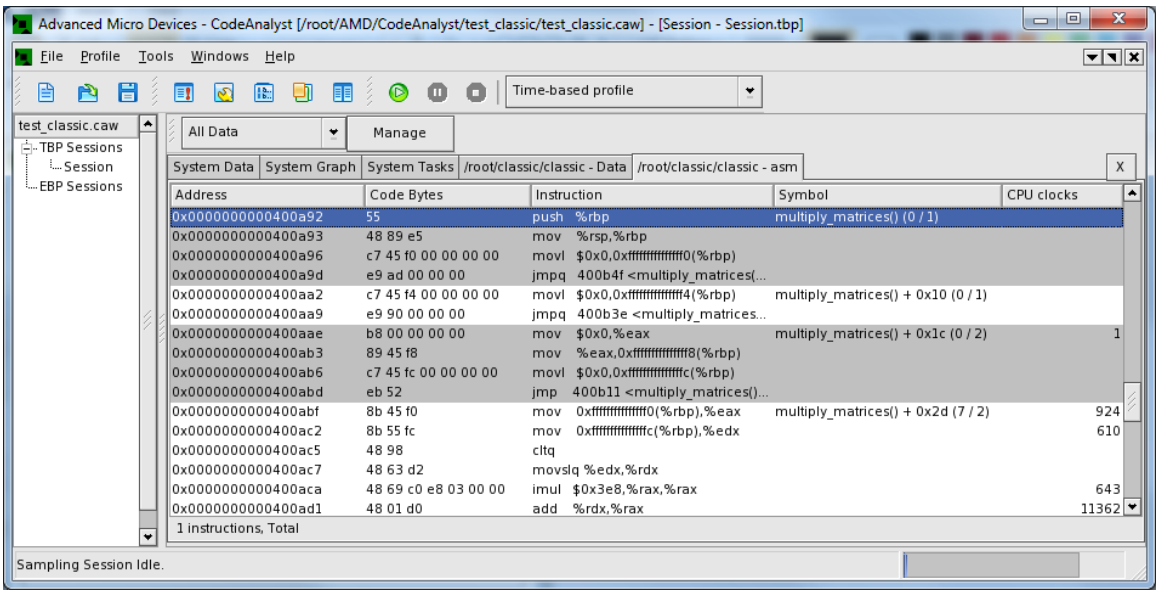
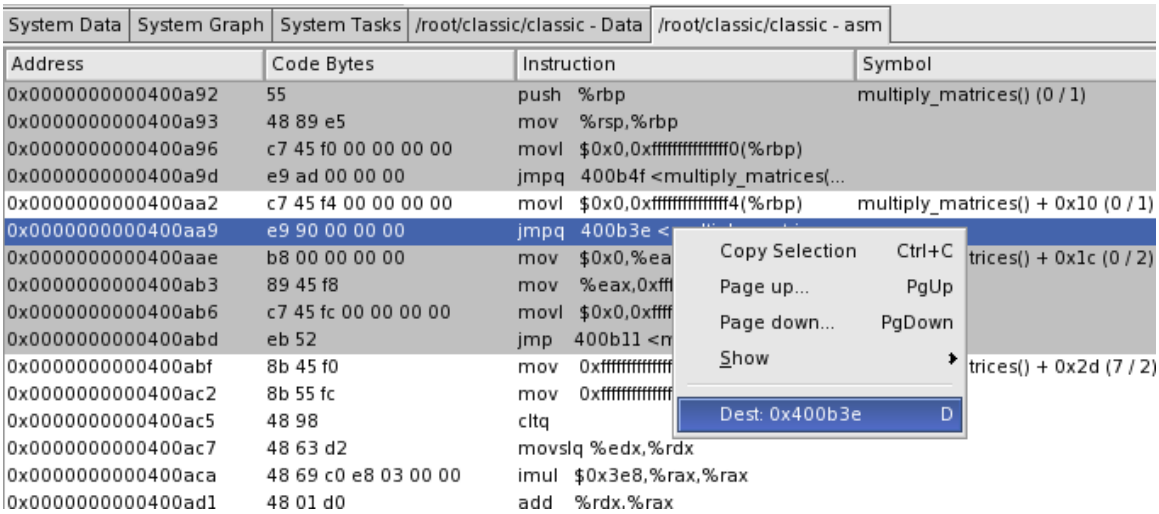


Figure 3.11. Basic block pop-up menu



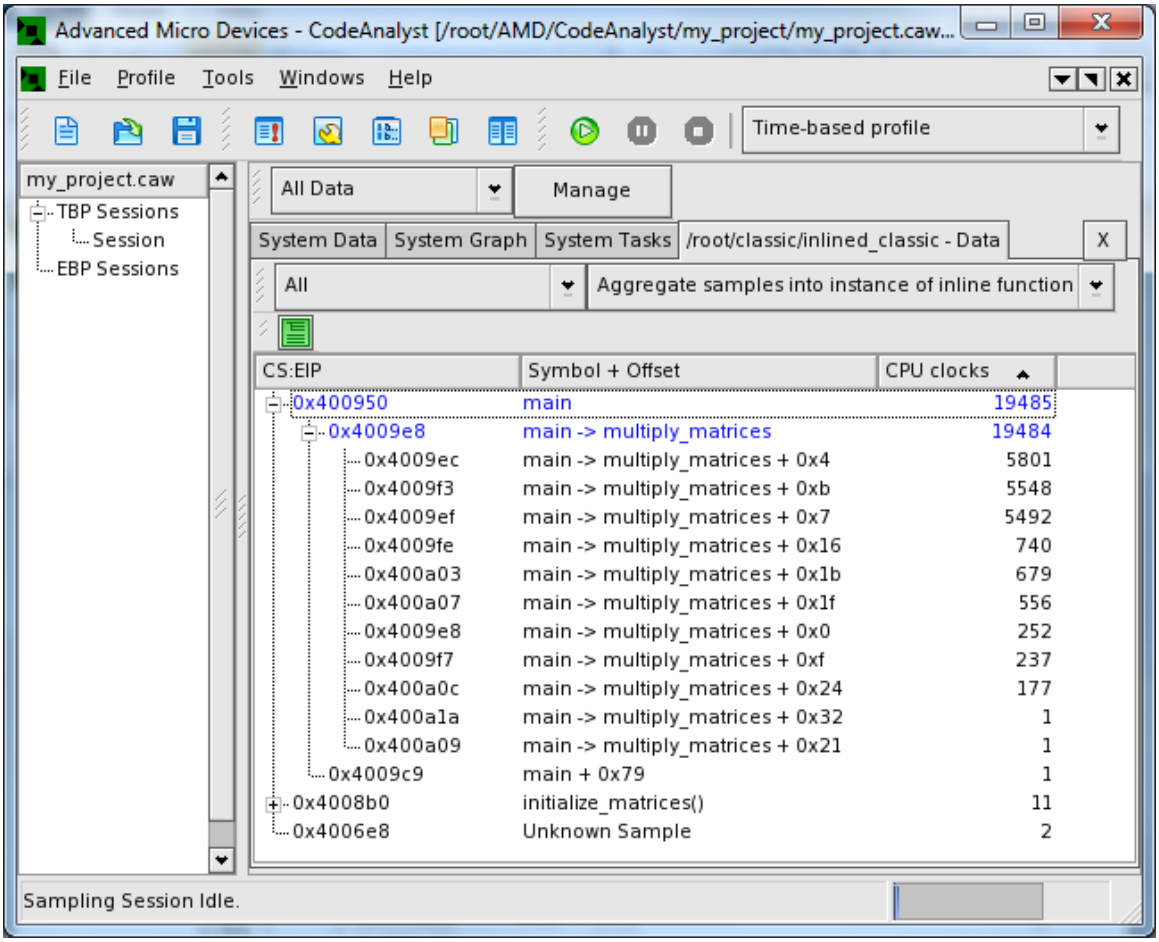
3.6. In-Line Analysis

CodeAnalyst provides two modes to help analyze in-line functions.

3.6.1. Aggregate samples into in-line instance

This is the default aggregation mode. Inline instance typically resides in the caller function. When samples belong to an in-line instance, CodeAnalyst aggregates them into the caller function and uses blue text to identify the in-line instance together with in-line function name. In the example in the previous image, **multiply_matrices** is an in-line function called by **main**. This mode aggregates samples belonging to the instance of **multiply_matrices** as part of **main**.

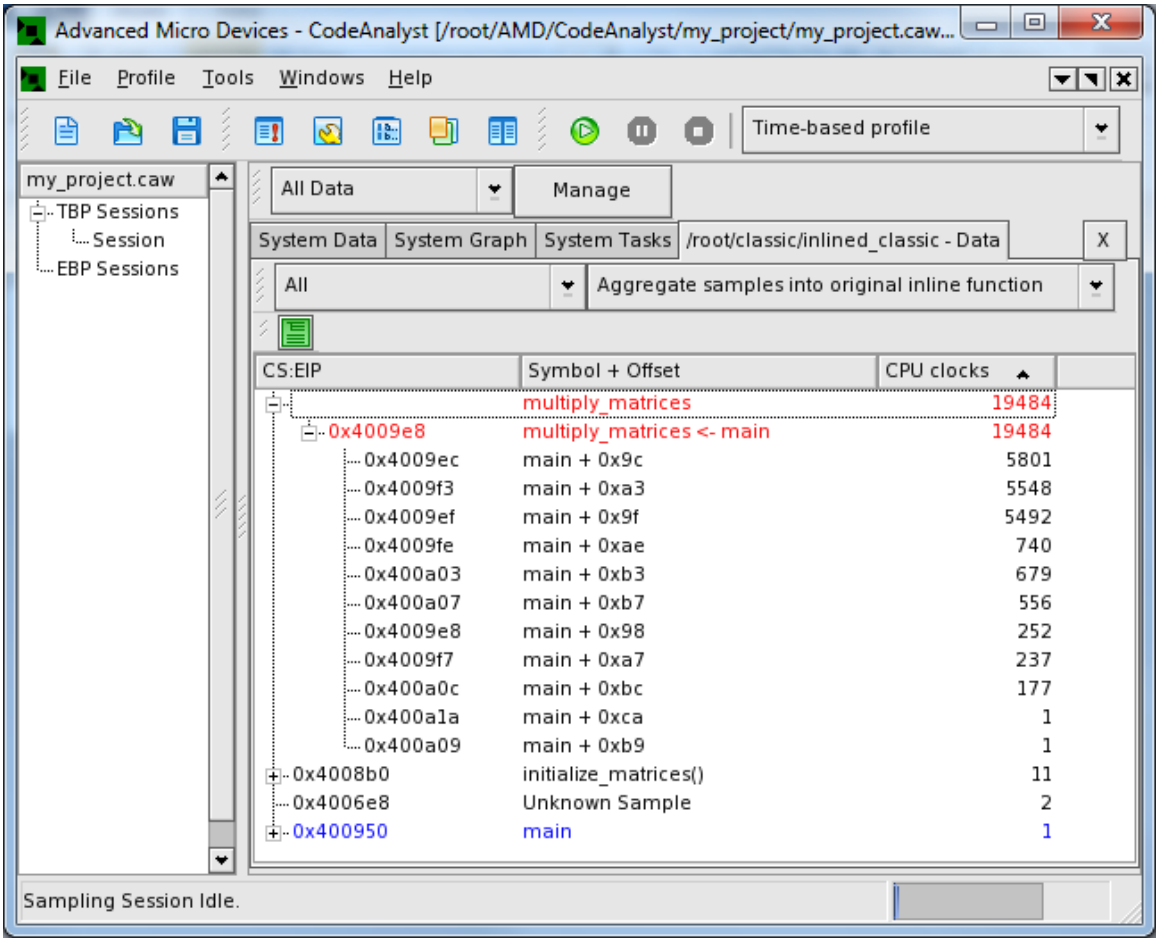
Figure 3.12. Aggregate into in-line instance



3.6.2. Aggregate samples into original in-line function

When samples belong to an inline instance, CodeAnalyst aggregates them into each inline instance. CodeAnalyst groups all inline instances and lists them together under the inline function which is presented in red text. In figure above, the inline function **multiply_matrices** has one inline instance in function **main**.

Figure 3.13. Aggregate into in-line function

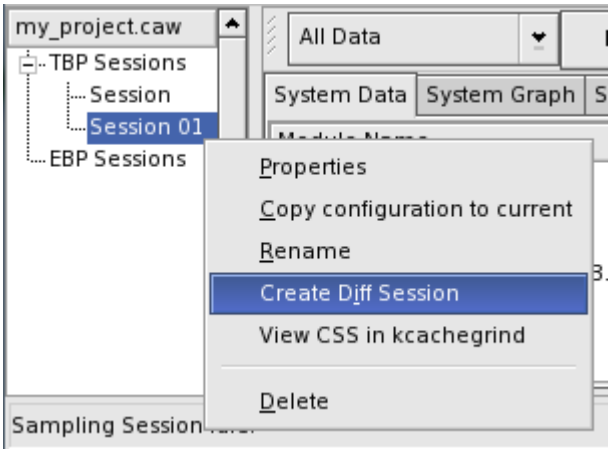


3.7. Session Diff Analysis

The AMD DiffAnalyst tool is a profile comparison (diff) tool that compares any two profiles, collected using theAMD CodeAnalyst tool. The AMD DiffAnalyst tool is distributed as part of the CodeAnalyst performance analyst and tuning suite. AMD DiffAnalyst is designed to help identify performance differences of any two binaries (i.e., executables or libraries). Users can compare performance data starting from the module level all the way down into each function and disassembly instruction. Please see *AMD DiffAnalyst for Linux*®, order# 45919, for more detail.

CodeAnalyst provides an interface to DiffAnalyst that allows users to differentiate (diff) any two sessions from the current project. Users can open diff session dialog by:

- Use the toolbar icon  , or
- Use the menu bar and select **Tools > Create Diff Session**, or
- Select the option, right-click, and from the menu select any session listed in the project navigation pane.



Use the following steps to create a diff session:

1. In the New Diff Session dialog, select from the list of available sessions in the current project.

2. Select two binaries to "diff" from the lists of **Available Tasks** and **Available Modules**.

3. Once binaries are selected, select **Create Diff Session**. The AMD CodeAnalyst tool invokes the AMD DiffAnalyst application with the selected configuration (see following example).

Figure 3.14. New Diff Session Dialog

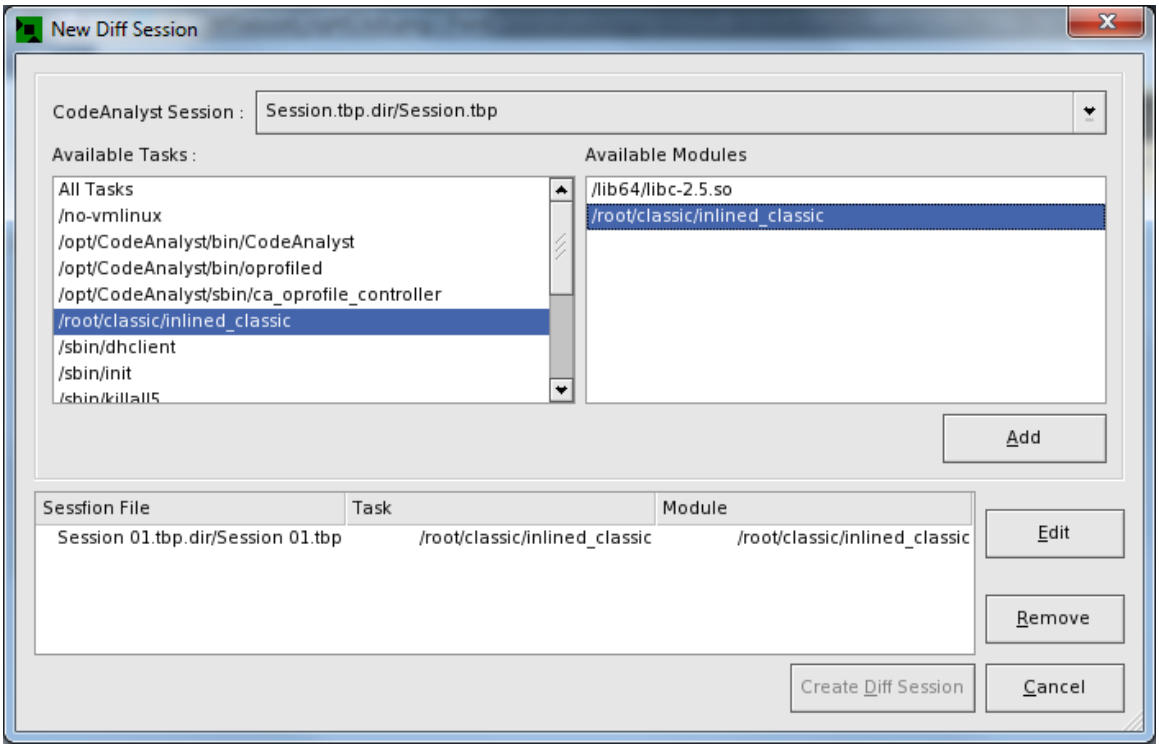
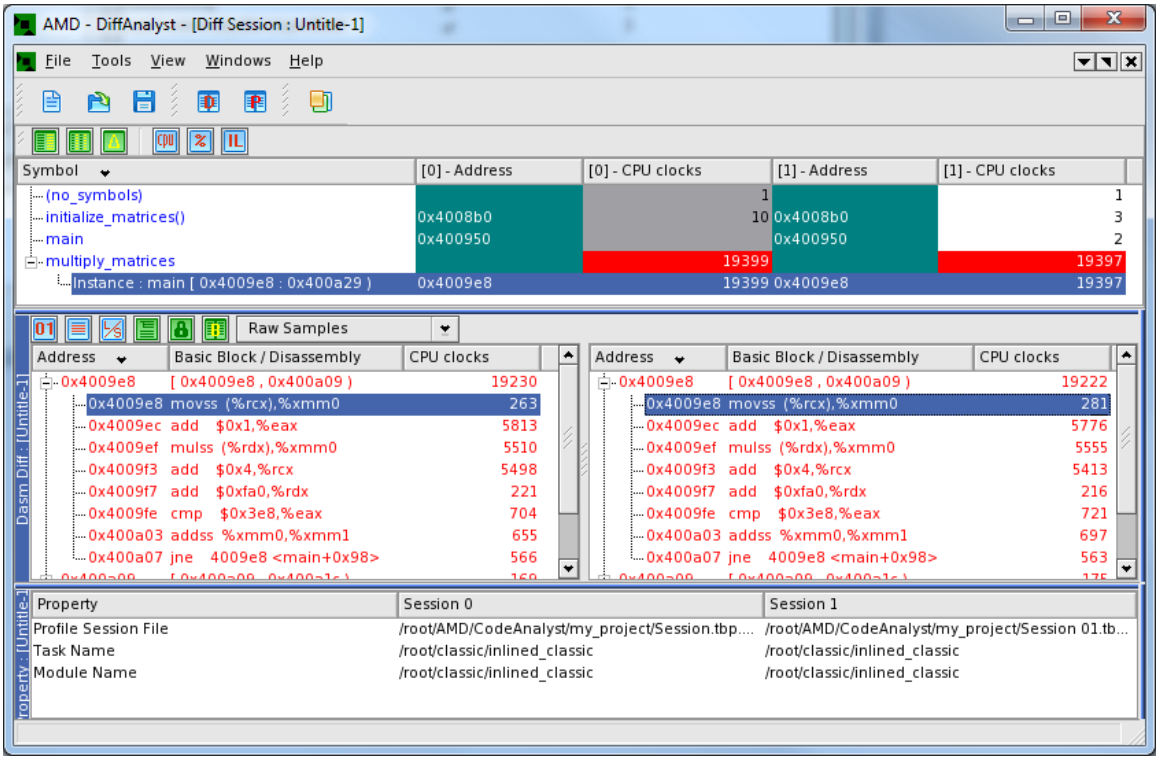


Figure 3.15. An example of the active AMD DiffAnalyst application.

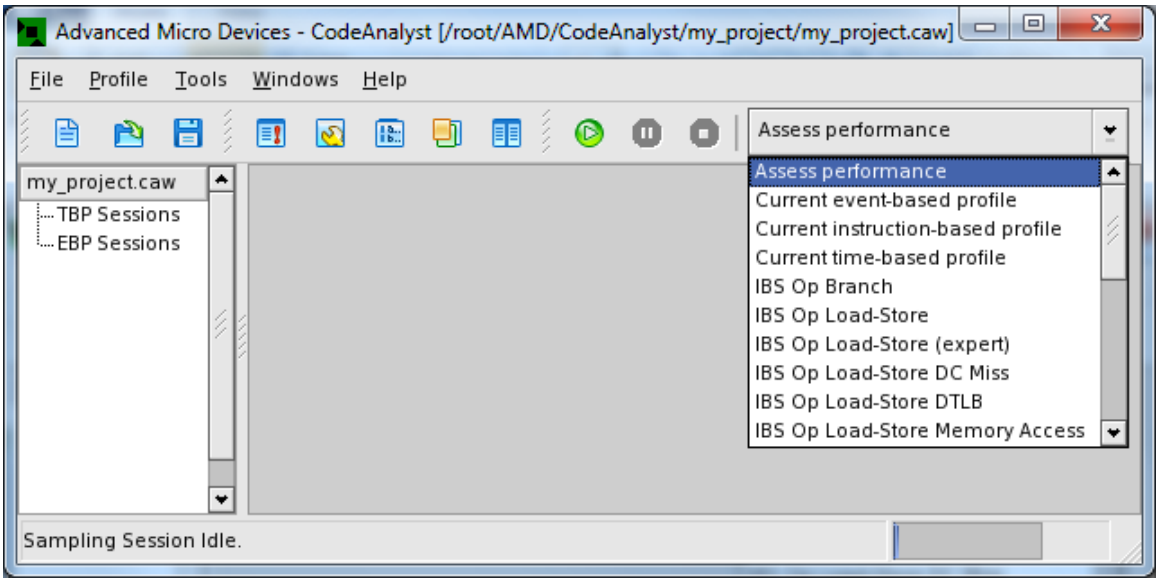


Chapter 4. Configure Profile

4.1. Profile Data Collection

Profile configurations control the kind of performance data that is collected during a session. CodeAnalyst provides several Section 4.4, “Predefined Profile Configurations” that support the most common kinds of performance analysis. Profile configurations eliminate the tedium of configuring data collection by hand for each session.

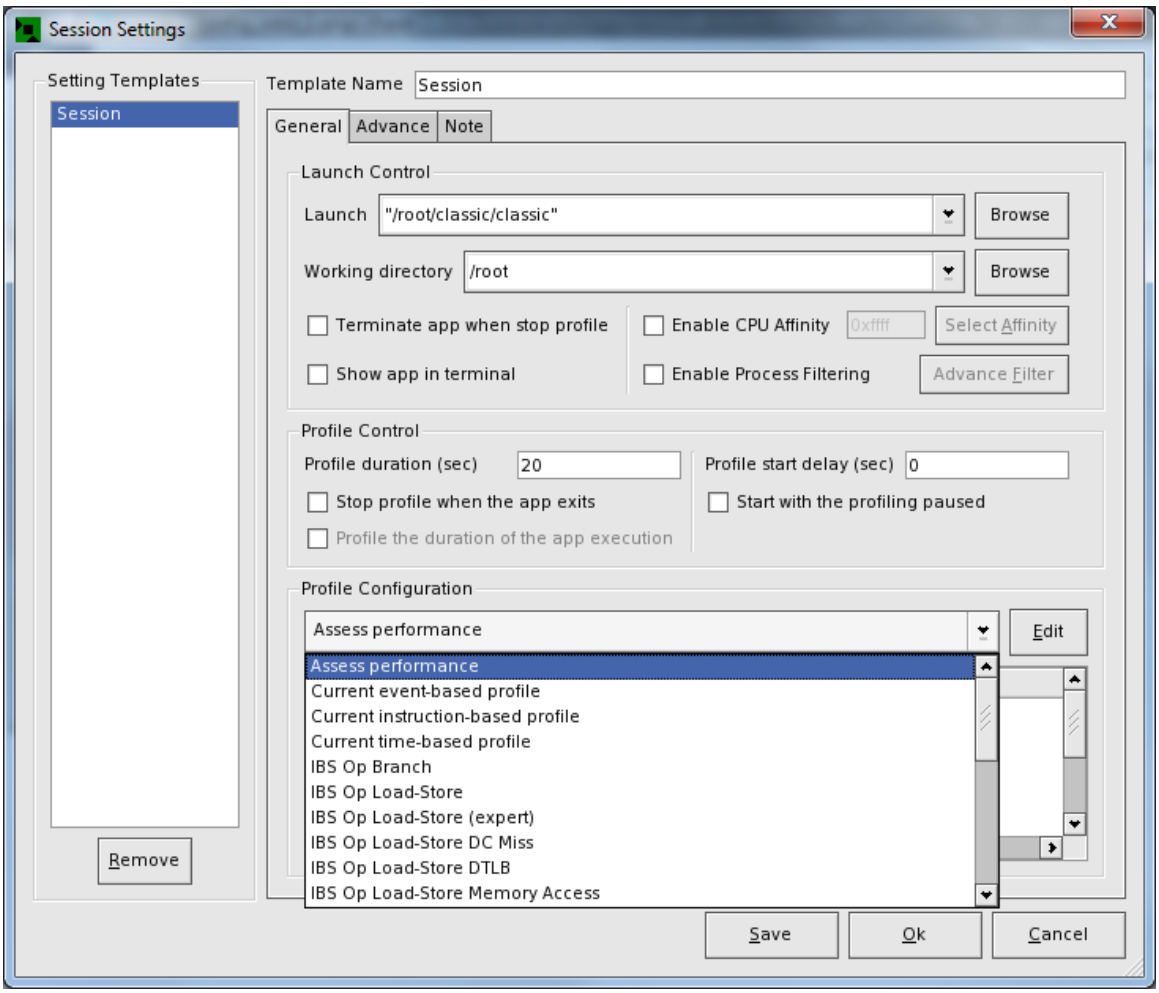
To quickly change to a different profile configuration, select the profile configuration from the drop-down list of program configurations in the toolbar.



To select a profile configuration via Session Settings:

1. Open the **Session Settings** dialog box by clicking the session settings button  in the toolbar, or by selecting **Tools > Session Settings** from the menu bar.

2. Select one of the profile configurations from the drop-down list of profile configurations in the Session Settings dialog box.



4.1.1. Modifying a Profile Configuration

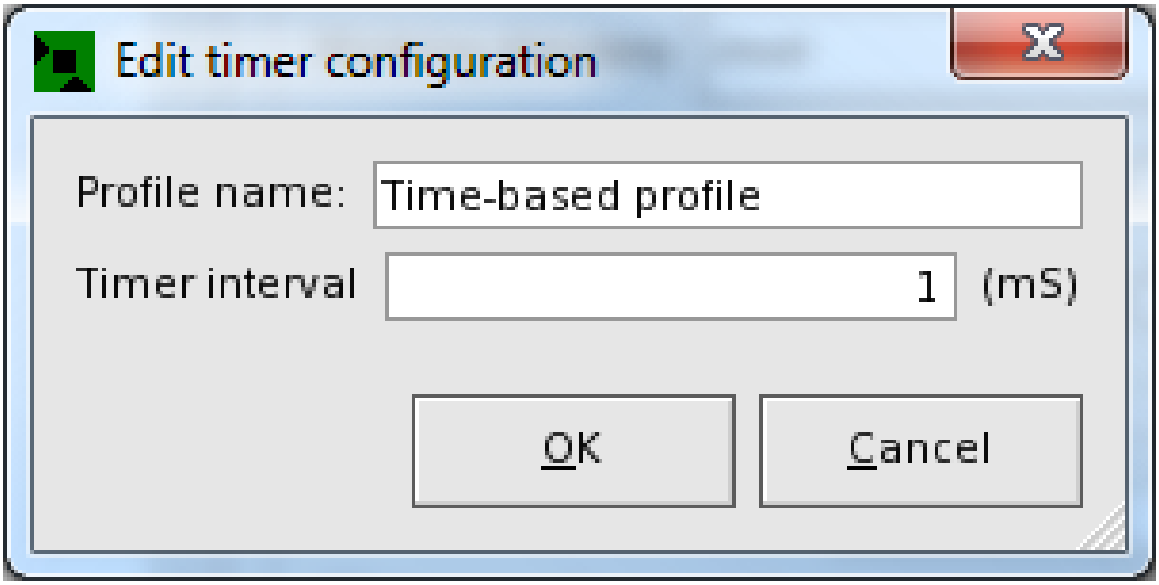
The predefined profile configurations should provide starting points for program analysis. However, eventually it is necessary to change either the kind of data collected (such as the specific events collected through event-based profiling), the sampling period, or some other aspect of data collection. Custom profile configurations can be created, modified and saved through **Configuration Management**. See Section 4.5, “Manage Profile Configurations” for further details.

4.2. Edit Timer Configuration

The Current time-based profile configuration opens a dialog box for changing:

- The profile configuration name.
- The timer interval that determines how often samples are taken.

Figure 4.1. Edit timer configuration



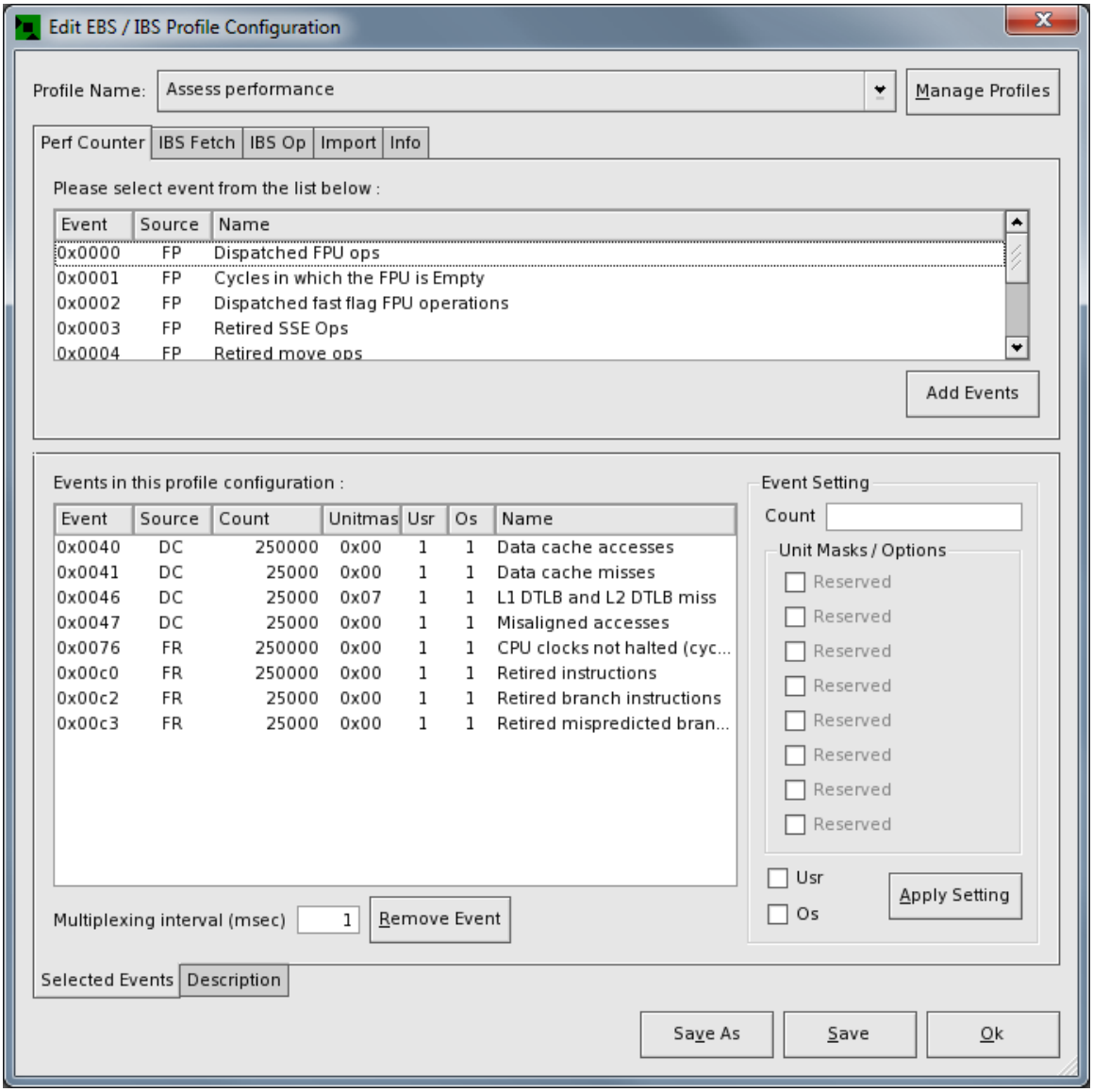
4.3. Edit Event-based and Instruction-based Sampling Configuration

This section discusses how to configure event-based and instruction-based sampling profile. The "Edit EBS/IBS Profile Configuration" dialog box allows for changing a profile configuration that uses event-based sampling and/or instruction-based sampling for data collection. This dialog corresponds to the dialog box opened by using the "**Edit**" button displayed in "**Session Settings** "dialog box. This dialog box allows for defining performance events to be measured.

It is important to understand the difference between event-based sampling and instruction-based sampling as they use different data collection techniques and different performance monitoring hardware. For more technical detail about event-based Sampling and instruction-based Sampling, please see Section 3.3, “Event-Based Profiling Analysis” and Section 3.4, “Instruction-Based Sampling Analysis”.

AMD processors can measure a wide range of event types. These events are described in more detail in the section Section 9.1, “Performance Monitoring Events (PME)”. Consult the **"BIOS and Kernel Developer's Guide" (BKDG)** for the AMD processor in your test platform for in-depth information about the performance monitoring events that it supports, including any processor revision-by-revision differences. For IBS-derived events, please see Section 9.3, “Instruction-Based Sampling Derived Events”

Figure 4.2. Edit EBS/IBS configuration



4.3.1. Profile Name

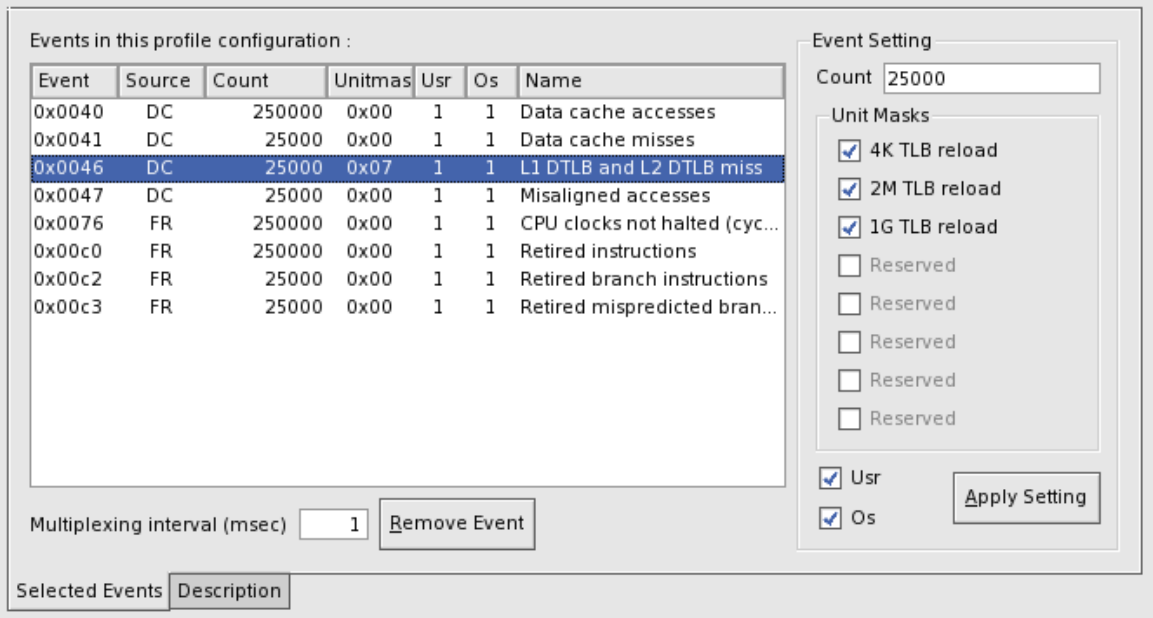
This drop-down menu shows the name of currently selected profile configuration for editing. Changing selection will re-populate settings with the newly selected configuration. Once changes are made to the current configuration, users must click "Ok" or "Save" to store changes to the current configuration, or "Save As" to store changes as a new configuration.

Note that the pre-defined configurations cannot be modified, but users can save the changes as a new user-defined configuration. Users can manage the list of configurations by clicking the "Manage Profiles" button which will bring up the "Configuration Management" dialog (See Section 4.5, “Manage Profile Configurations” for more info).

4.3.2. Select and Modify Events in Profile Configuration

4.3.2.1. Selected Events Tab

Figure 4.3. Edit EBS/IBS configuration: Selected Events Tab



This tab contains a table listing performance events in the currently selected profile configuration. For instance, when you select "Assess Performance" profile configuration, there will be 8 events shown in the table along with the counts, unitmask, and other informations. To modify an event, simply highlight the event you want to modify. Setting of the selected event will get populated into the "Event Setting" field on the right.

- Count - The "Count" field specifies the event count (sampling period) for an event.
- Unitmasks / Options - Check boxes allow for specifying the unit mask or option for the selected event.
- Usr - Selecting **"Usr"** enables collection of user-level samples for an event.
- Os - Selecting **"Os"** enables collection of operating system-level samples for an event.

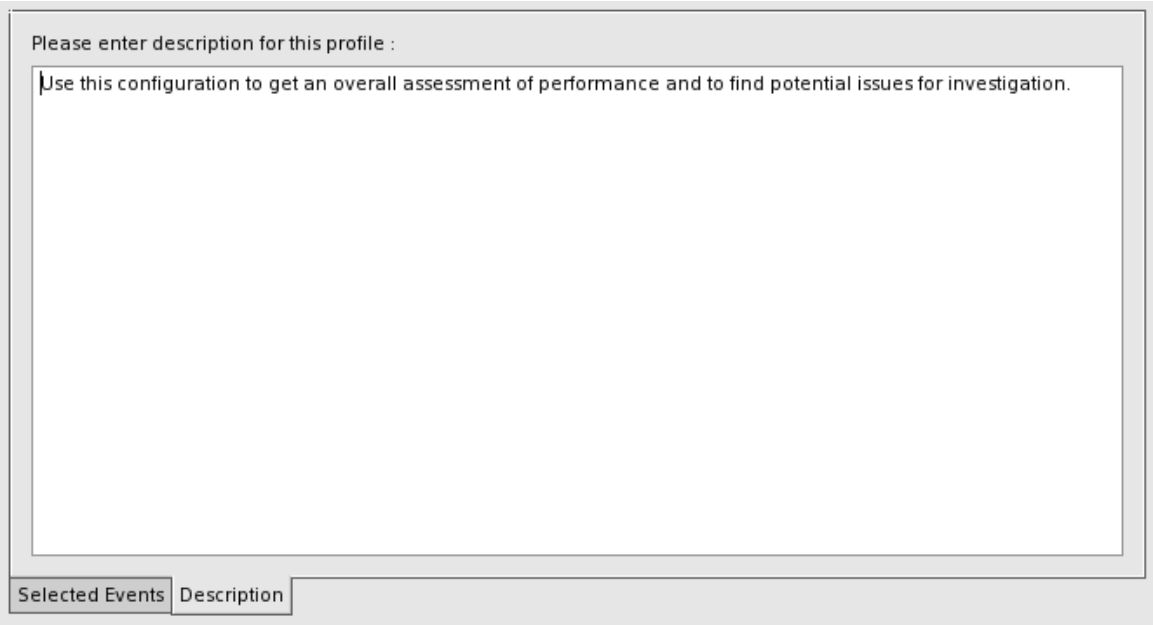
Users can make desire changes, and then click "Apply Setting" button to save changes.

On an AMD processor, there are limited number of performance event counter hardware. In case the number of events selected is more than the number of available hardware, performance counter event multiplexing will be enabled to help eliminate the hardware limitation. Here, the "Multiplxing Interval" field will be enabled and default to 1 msec. This parameter basically specifies the profiling period before the next group of events is multiplexed during profile run; as described in Section 2.4, “Event Counter Multiplexing”.

To remove an event, simply select the event and click "Remove Event" button.

4.3.2.2. Description Tab

Figure 4.4. Edit EBS/IBS configuration: Description Tab



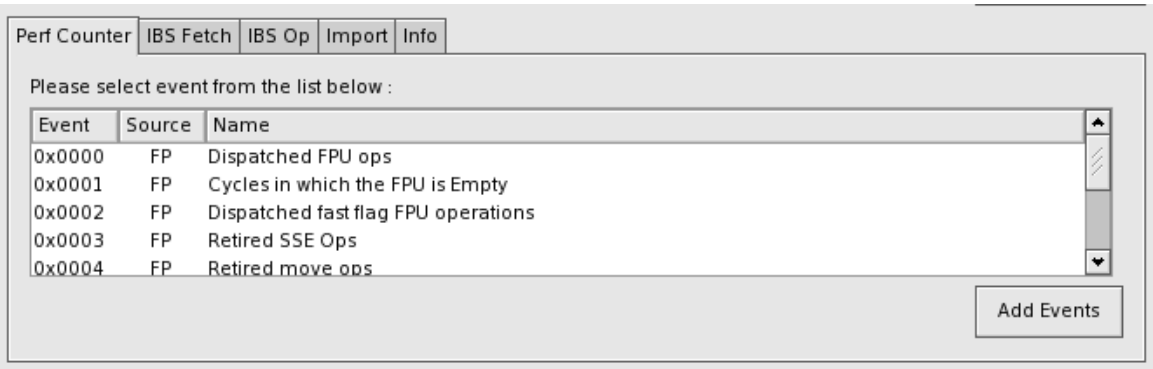
This tab contains a field which let users describe the purpose of the current profile configuration.

4.3.3. Available Performance Events

These sections describe how to add performance events to the current profile configuration. Users can select any type of event to add to the configuration (i.e. EBS event and IBS-derived events can be added in a configuration).

4.3.3.1. Perf Counter Tab

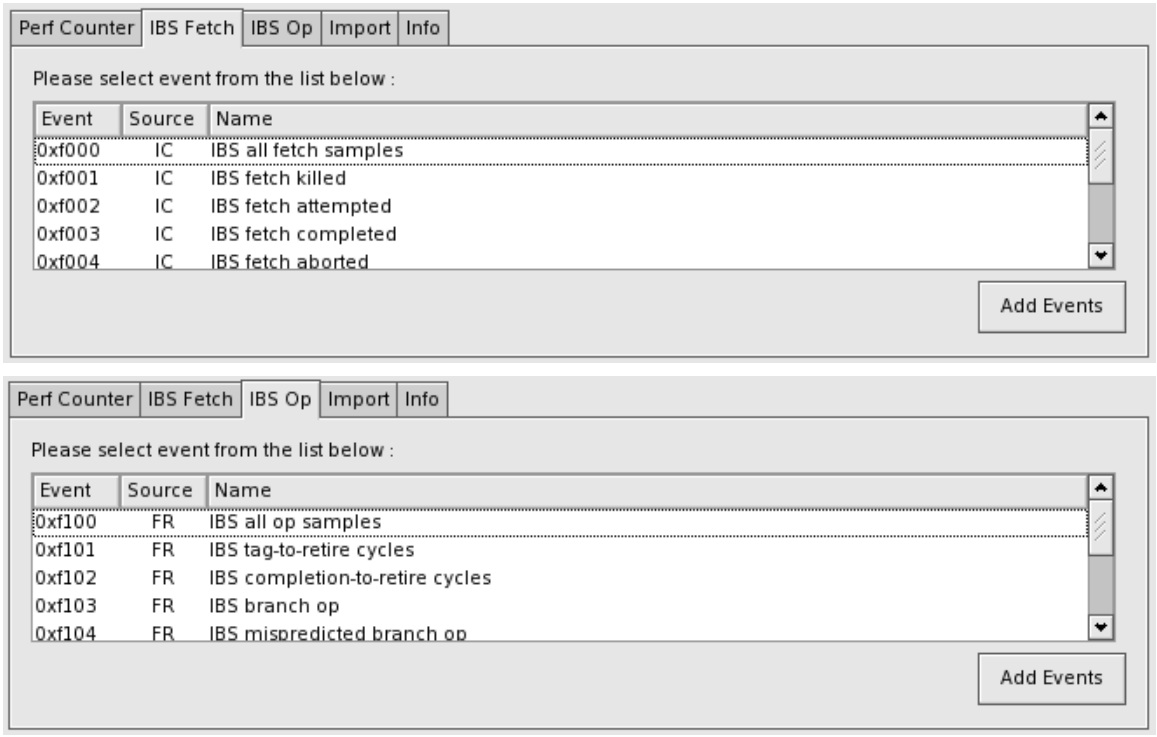
Figure 4.5. Edit EBS/IBS configuration: Perf Counter Tab



This tab contains the list of Event-based Sampling performance events. Only performance events available for the currently running system are shown. Users can select multiple events and click "Add Event" to add them to the current profile configuration.

4.3.3.2. IBS Fetch / Op Tab

Figure 4.6. Edit EBS/IBS configuration: IBS Fetch / Op Tab

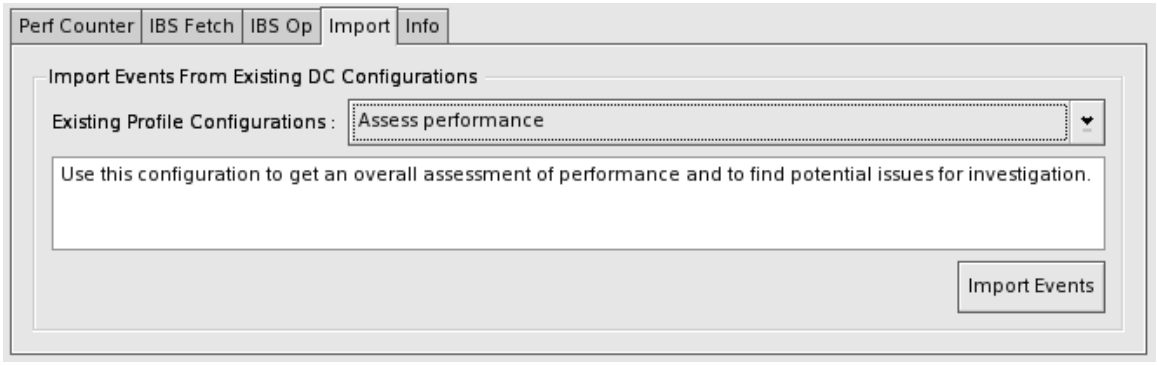


These tabs contain the list of IBS-Fetch and IBS-Op derived performance events. These tabs are available only if the currently running system supports this performance monitoring feature. Users can select multiple events and click "Add Event" to add them to the current profile configuration.

NOTE: All selected IBS-Fetch derived events must have same settings (i.e. same counts and option-mask). The same rule applies to IBS-Op.

4.3.3.3. Import Tab

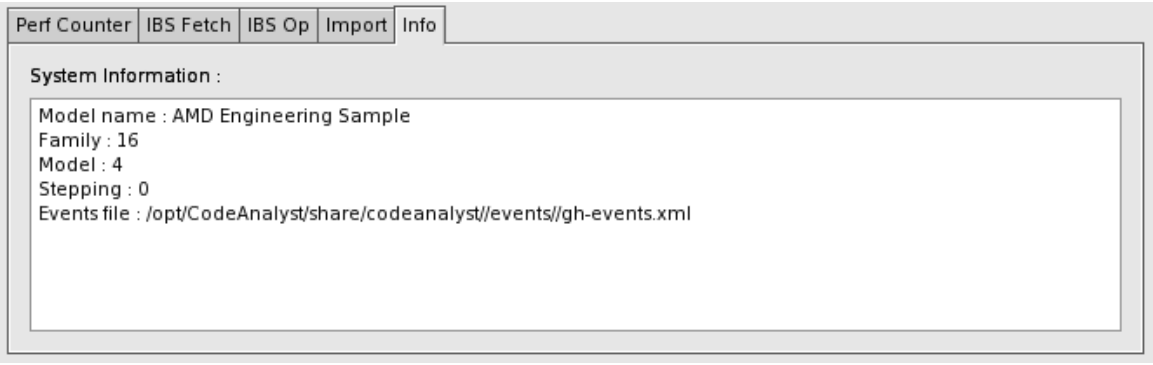
Figure 4.7. Edit EBS/IBS configuration: Import Tab



Besides selecting individual events, users can import event selections from an existing profile configuration. This tab lists the available profile configuration in the drop-down menu. Once you select a configuration, description of the configuration will be shown in the field below. To add events, simply click on "Import Events".

4.3.3.4. Info Tab

Figure 4.8. Edit EBS/IBS configuration: Info Tab



This tab shows information of the currently running system such as processor family, model, etc. It also shows the XML events file currently in use.

4.4. Predefined Profile Configurations

AMD processors offer a wide range of event types for collection and this can overwhelm new users. Predefined profile configurations address the more common aspects of program and system performance analysis. CodeAnalyst provides configurations that support various methods of data collection including time-based profiling (TBP, event-based profiling (EBP), and Instruction-Based Sampling (IBS).

Profile Configuration	**Purpose**
Time-based profile (TBS)	Collect a time-based profile
Assess performance (EBS)	Collect a profile that provides an overall assessment of performance
Investigate data access (EBS)	Investigate data cache (DC) and data translation lookaside buffer (DTLB) performance
Investigate instruction access (EBS)	Investigate instruction cache (IC) and instruction translation lookaside buffer (ITLB) performance
Investigate L2 cache access (EBS)	Investigate access to the unified L2 cache
Investigate branching (EBS)	Investigate branch behavior including branch misprediction
Instruction-based sampling (IBS)	Investigate instruction fetch and macro-op execution performance
IBS Op Overall Assessment	Get an overall assessment of performance using IBS Op
IBS Op Branch	Find poorly predicted branches
IBS Op Load-Store	Measure data load / store performance
IBS Op Load-Store (expert)	Measure data load / store performance (expert)
IBS Op Load-Store DC Miss	measure data load / store DC miss performance
IBS Op Load-Store DTLB	Measure data load / store DTLB performance
IBS Op Load-Store Memory Access	Measure data load / store memory access performance
IBS Op Load-Store Page Size	Measure data load / store page size performance

Profile Configuration	Purpose
IBS Op Northbridge Access	Measure northbridge access performance
IBS Op Northbridge Cache Access	Measure northbridge cache access performance
IBS Op Northbridge Services	Measure northbridge services performance
IBS Op Return	Find poorly predicted returns

The listed configurations are not modifiable. However, CodeAnalyst provides three modifiable profile configurations which can be used as templates for creating customized profile configurations:

- Current time-based profile
- Current event-based profile
- Current instruction-based profile

4.5. Manage Profile Configurations

CodeAnalyst calls the process of creating, modifying and saving a custom, user-defined profile configuration "**Configuration Management**." The Section 4.4, “Predefined Profile Configurations” are preset and cannot be changed. However, a profile configuration can be saved under a different name and the new profile configuration can be modified. Three modifiable profile configurations are also available:

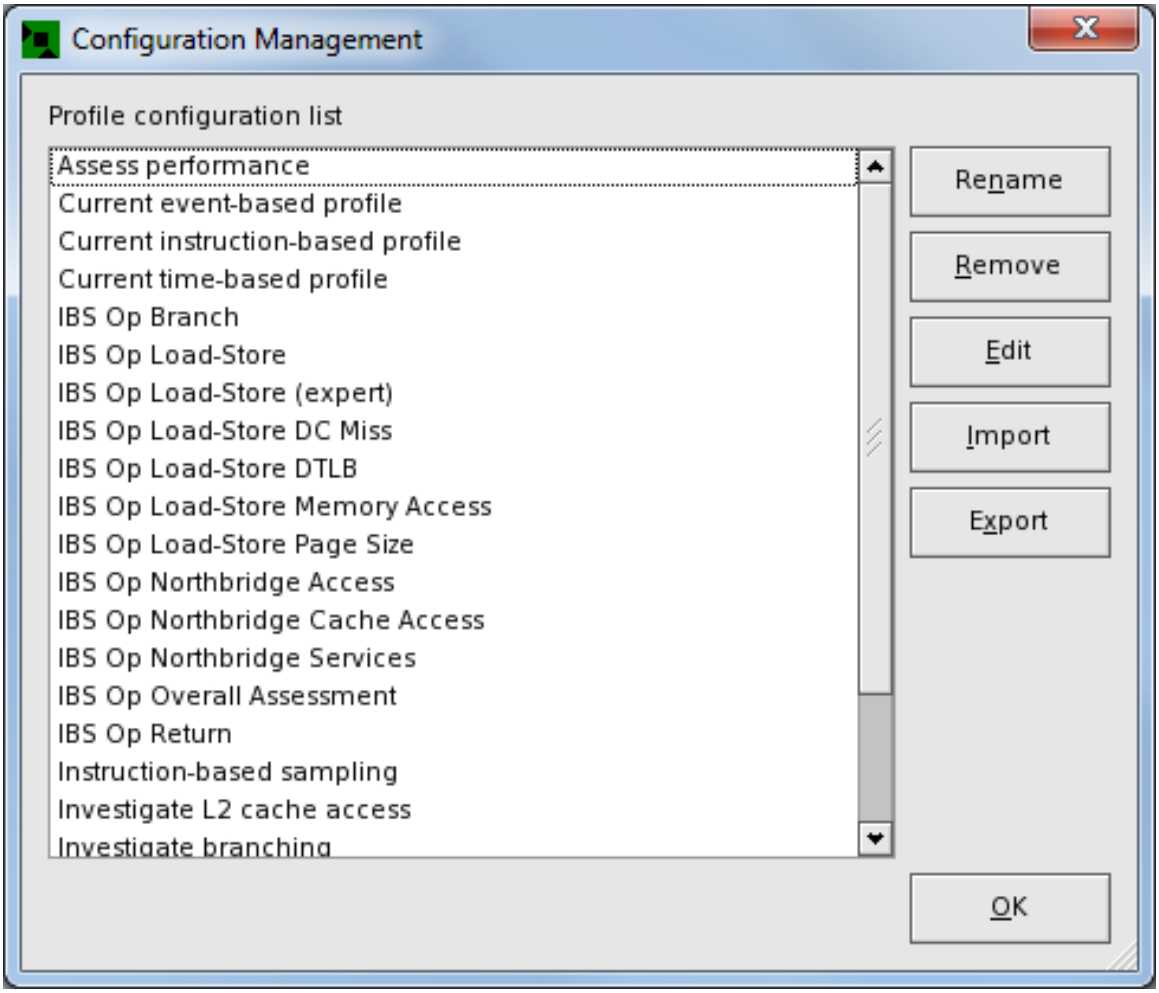
- Current time-based profile
- Current event-based profile
- Current instruction-based profile

These profile configurations act as templates for creating new user-defined profile configurations. The "Current" profile configurations are persistent and can be used as temporary "scratchpad" profile configurations. To start configuration management:

1. Open the Configuration Management dialog box by clicking on the Configuration Management button  in the toolbar or select **Tools > Configuration Management** from the menu bar.
2. The Configuration Management dialog box appears. Select a profile configuration in the **Profile configuration list**.
3. Click one of the configuration management buttons on the right hand side of the dialog box.

The Configuration Management dialog box selects a profile configuration to be modified (**Edit, Rename**), to be deleted (**Remove**), or to be written to a file (**Export**.) A new profile configuration can be read from a file (**Import**.) A profile configuration is stored as a sharable .XML file. Export writes an existing profile configuration to a file while import adds a new profile configuration by reading it from a file. New profile configurations appear at the bottom of the profile configuration list. Only user-defined and imported configurations can be deleted from the list; the predefined profile configurations that are installed with CodeAnalyst cannot be deleted.

Figure 4.9. Configuration Management



The following actions are available through the Configuration Management dialog box:

- Click the **Rename** button to rename the selected, user-defined profile configuration (e.g., **my config** configuration) and to rename the associated file. Only user-defined profile configurations may be renamed.

- Click the **Remove** button to remove the selected, user-defined profile configuration (e.g., **my config** configuration) and to delete the associated file. Only user-defined profile configurations may be removed from the list of profile configurations.

- Click the **Edit** button to modify the selected configuration.

- Click the **Import** button to import a profile configuration from an existing file. This adds a new profile configuration to the list. The import action displays a standard file selection dialog box. Use this dialog box to select a file to import.

- Click the **Export** button to write the selected profile configuration to a file. The export action displays a standard file selection dialog box. Use this dialog box to specify the name and location of the file to which the profile configuration is written.

- Click **OK** to close the window without taking further action.

The Edit button opens an edit dialog box. The kind of analysis used by the selected profile configuration determines the type of dialog box that displays since each kind of analysis has its own settings. For

example, selecting "Current time-based profile " then clicking the Edit button, opens the "Edit timer configuration" dialog box (See Section 4.2, “Edit Timer Configuration” and Section 4.3, “Edit Event-based and Instruction-based Sampling Configuration”). The user can then change the timer interval, which determines how often samples are taken.

Figure 4.10. Edit timer configuration

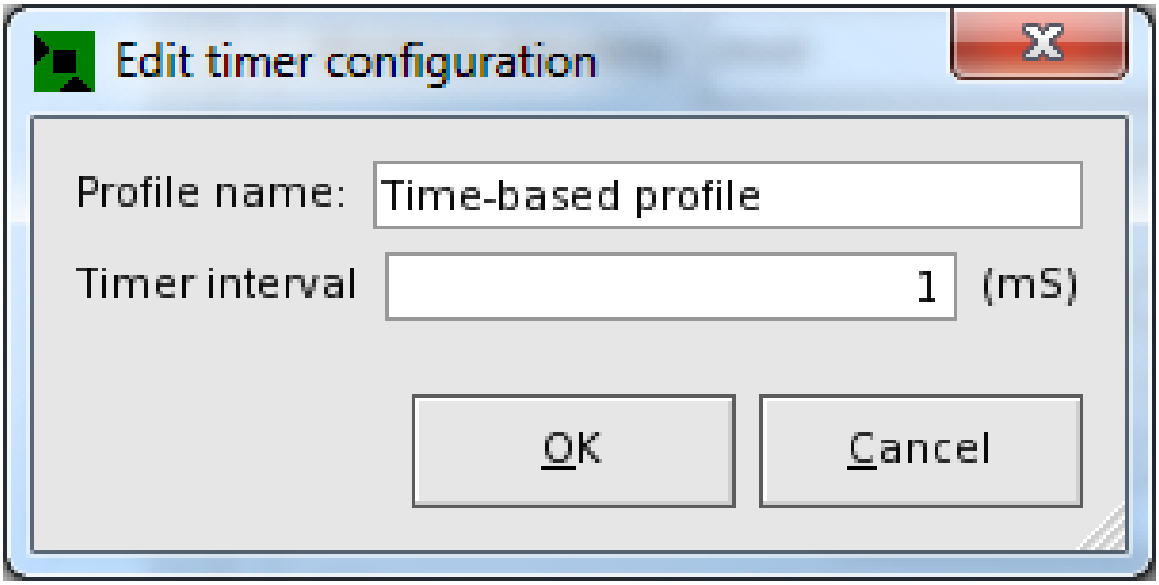
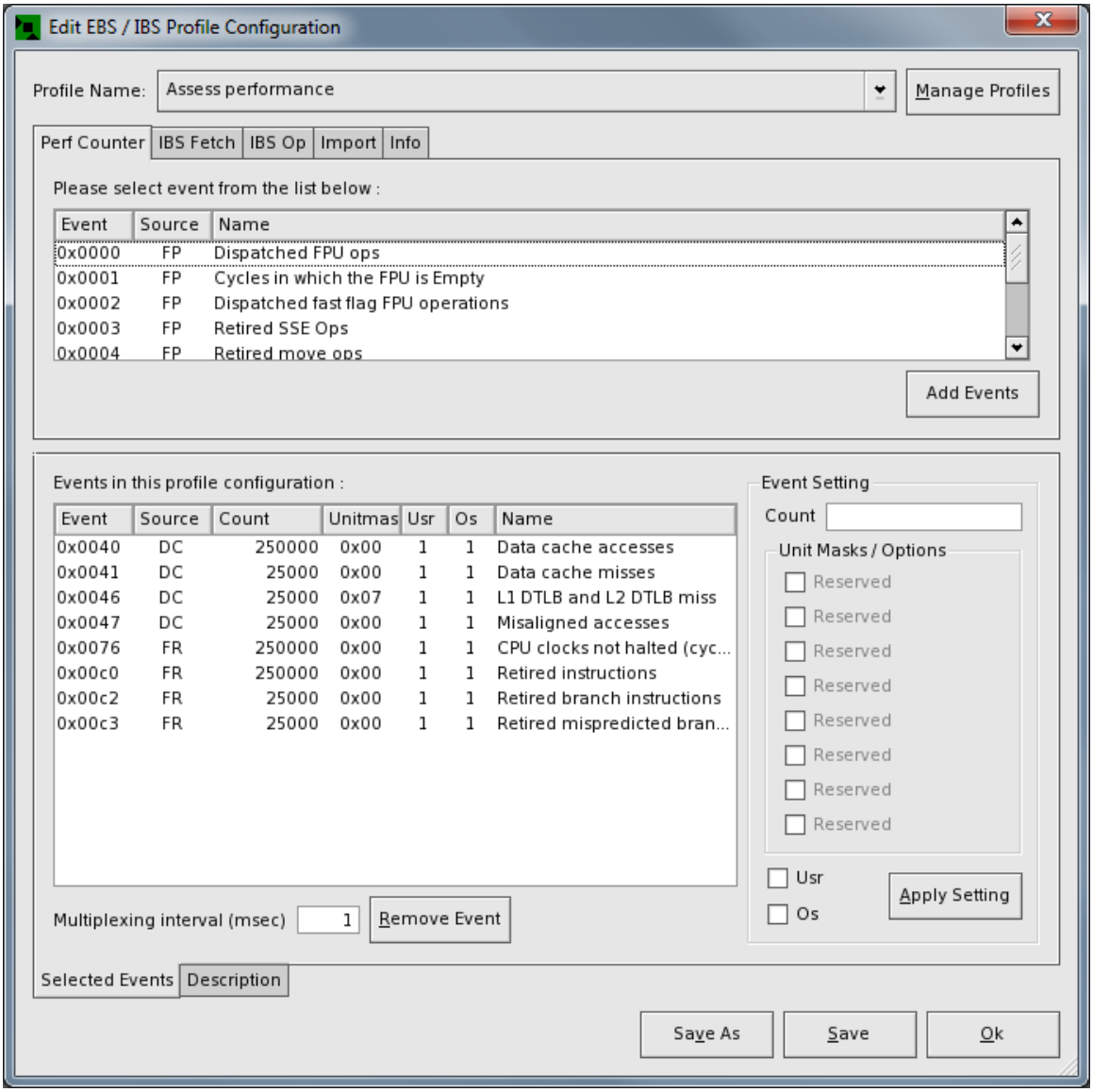


Figure 4.11. Edit events configuration



The **Session Settings** dialog (See Section 2.7, “Session Settings”) box contains an **Edit** button that also opens a profile configuration edit dialog box. Selecting a profile configuration and clicking the Edit button in the Session Settings dialog box is equivalent to opening the Configuration Management dialog box, selecting a profile configuration and clicking the Edit button.

By default, new profile configurations are stored in the **/$(HOME)/.CodeAnalyst/Configs/DC-Configs** subdirectory.

Chapter 5. Collecting Profile

5.1. Collecting Profiles and Performance Data

Information about program performance can be collected in several ways depending upon the kind of analysis to be performed. The following sections discuss ways to collect program profiles and performance information:

Section 5.2, “Collecting a Time-Based Profile”

Section 5.3, “Collecting an Event-Based Profile”

Section 5.4, “Collecting an Instruction-Based Sampling Profile”

Collection and analysis of profiles are also covered in the Chapter 8, *Tutorial*.

5.2. Collecting a Time-Based Profile

In time-based profiling, the application to be analyzed is run at full speed on the same machine that is running AMD CodeAnalyst. Time-based samples (collected at predetermined intervals) can be used to identify possible bottlenecks, execution penalties, or optimization opportunities. The timer-based profiling feature can be used with all AMD processors. This page describes how to collect a time-based profile.

The predefined profile (data collection) configuration (**Time-based profile**) is used to enable collection of a time-based profile.

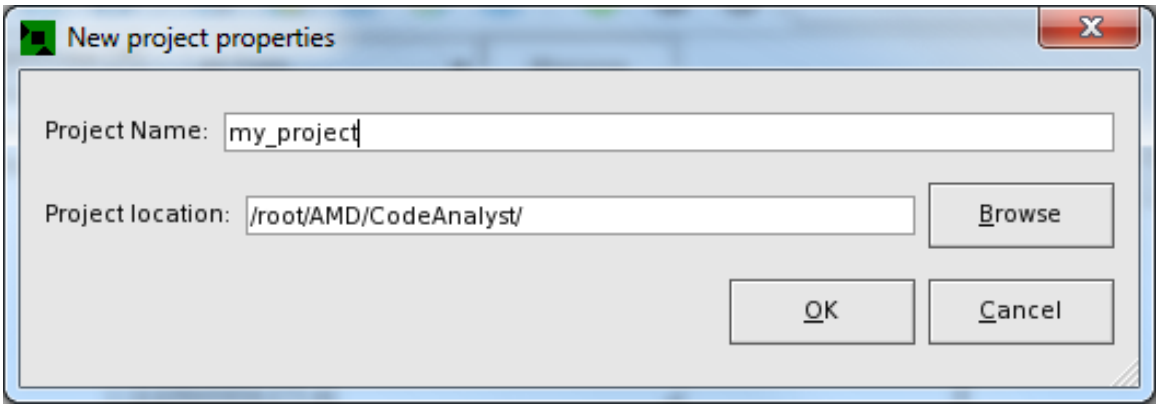
5.2.1. Collecting a Time-Based Profile

To collect a time-based profile :

1. Create new project or choose a previously opened project. Select **Time-Based Profile** from the drop-down profile configuration list in the toolbar.

2. If a new project is to be created, the **New project properties** dialog box opens. Assign a project name and location or browse for an existing file.

Figure 5.1. New Project Properties

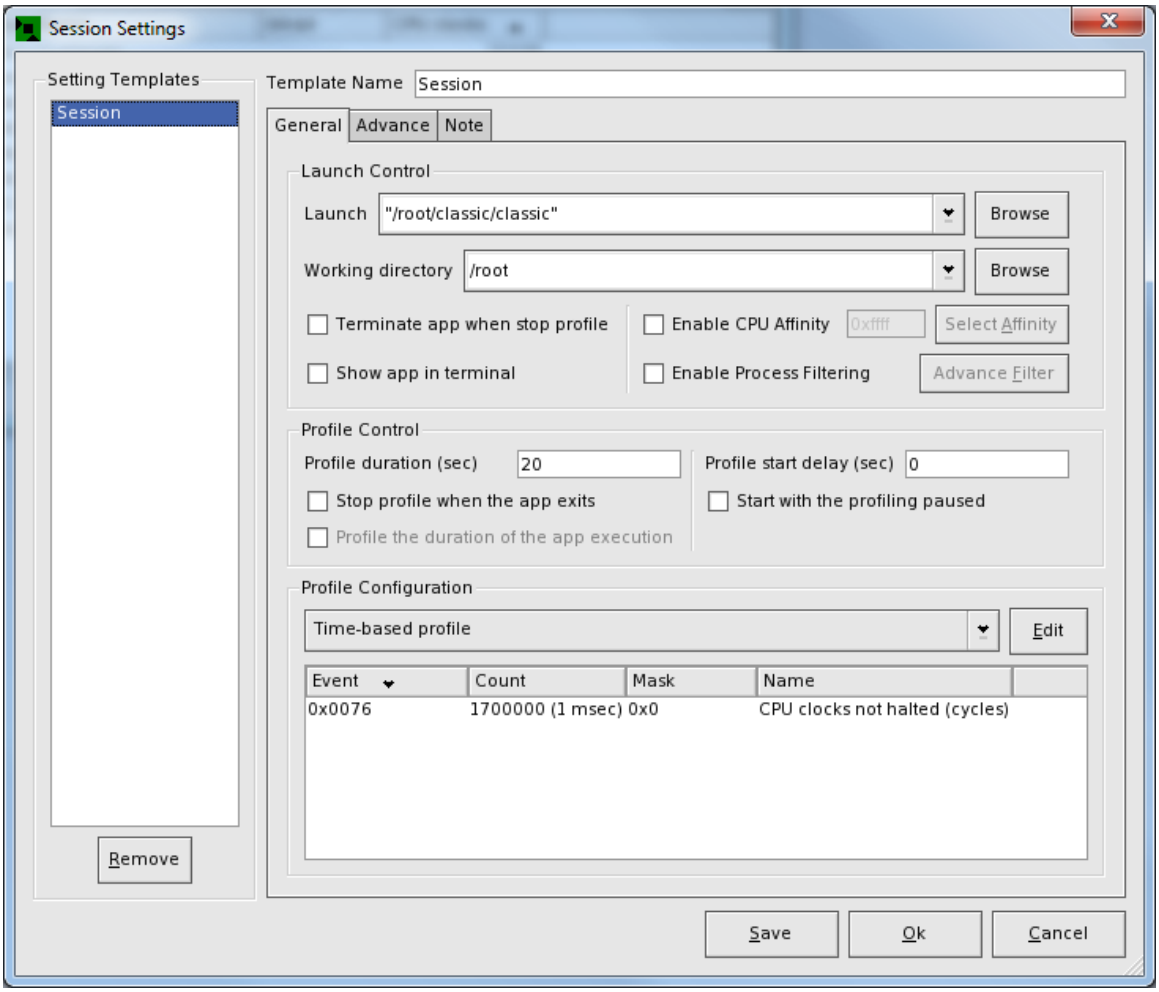


3. The **Session settings** window opens. Assign a session name (optional), enter the path to the application program to be launched and set the working directory.

4. Under **Profile configuration:** select time-based profile.

5. Select other desired profiling options and click OK.

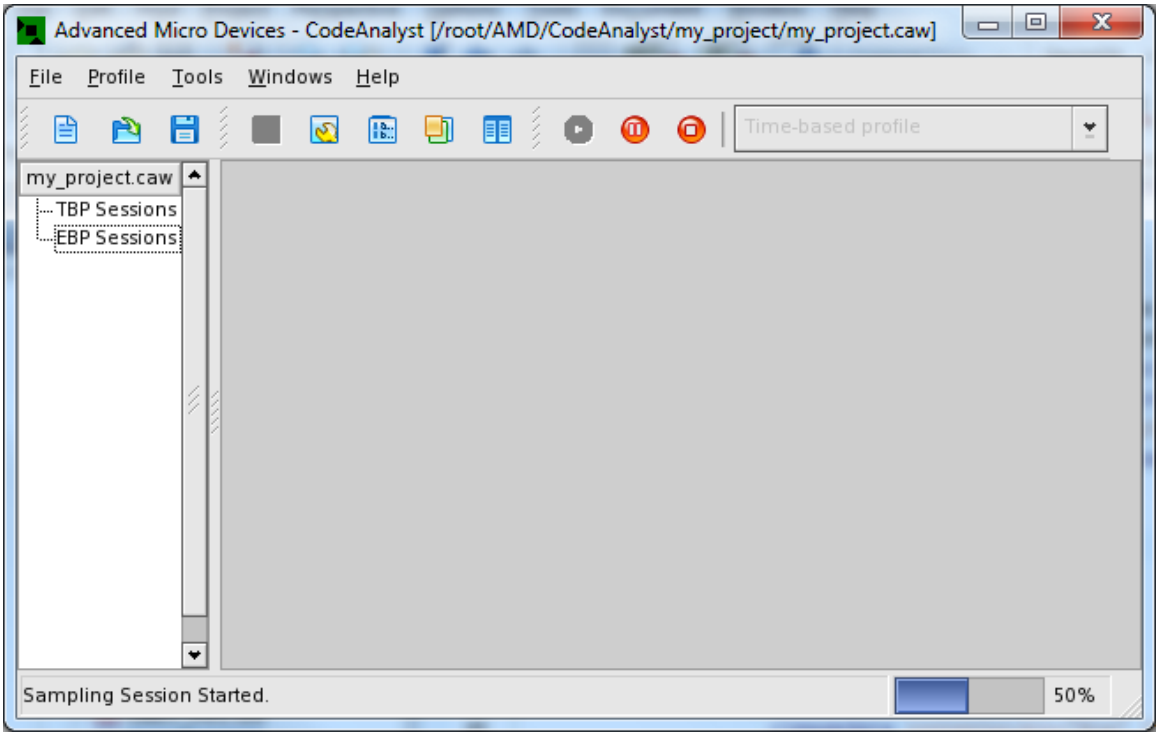
Figure 5.2. Session settings



6. Click the Start icon  to launch the application and begin profiling.

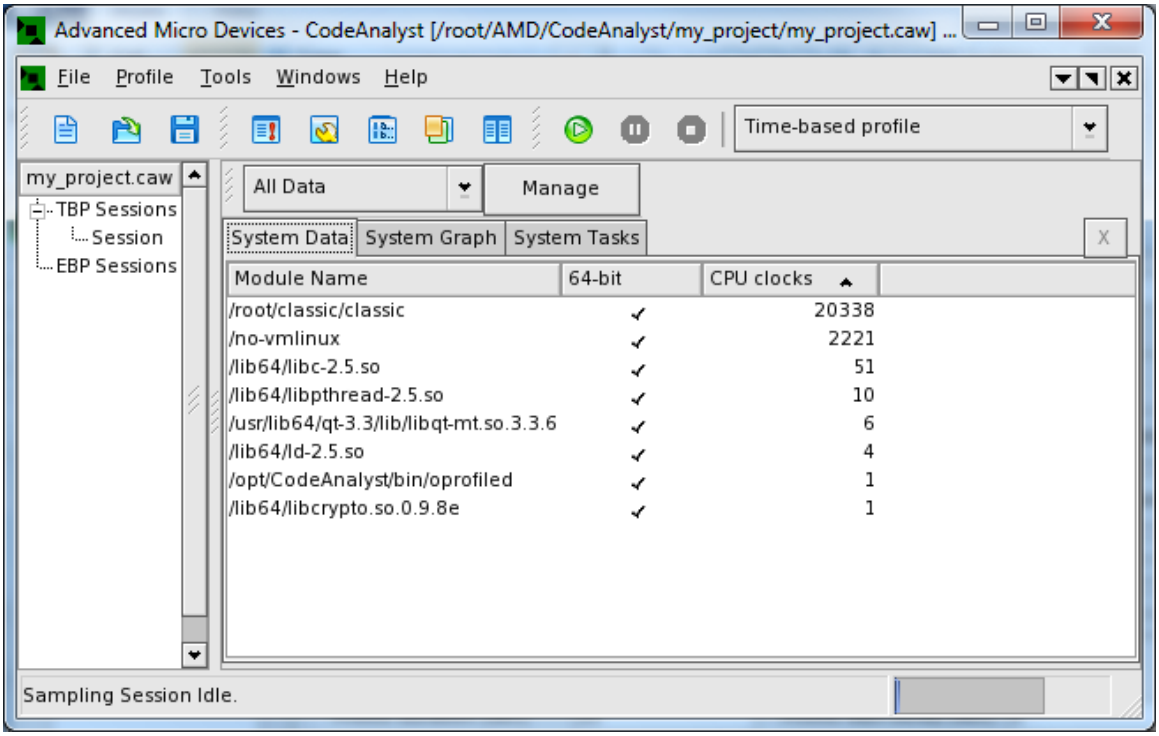
7. The task bar at the bottom of the screen displays "Sampling Session Started" and the percent completed. The **Pause** and **Stop** icons  become active.

Figure 5.3. Task Bar display



8. When the sampling session is complete, the application under test terminates and the performance data is processed. The work space then displays a module-by-module breakdown of the results in the System Data table. Select the System Graph tab to see the results in graphical form. Select the System Tasks tab to see a task-by-task breakdown of the results. Double-click on a module or task to drill down into the data.

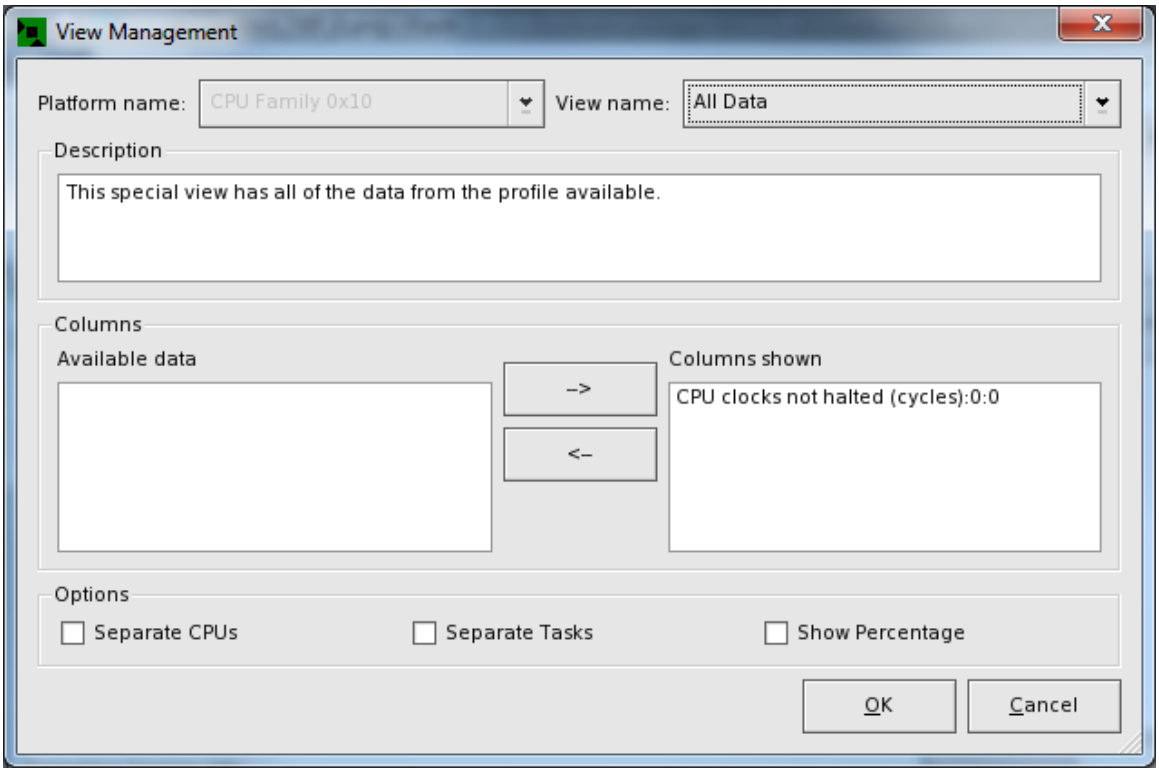
Figure 5.4. System Data results



5.2.2. Changing the Current View of the Data

To change the type of data displayed in the current view, click **Manage**. The **View Management** dialog box opens. Refer to the Section 7.3, “View Management” section for details. The items listed in the Columns part of the View Management dialog box depend on the view configuration that is in current use.

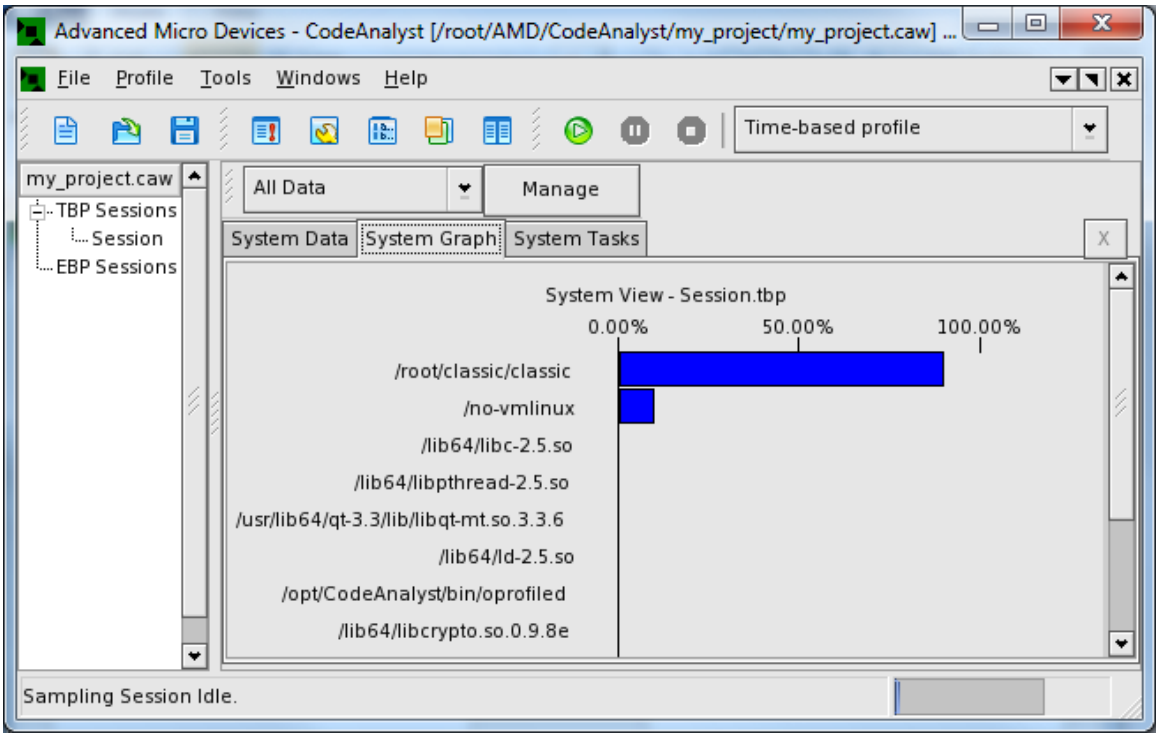
Figure 5.5. View Management



5.2.3. System Data and System Graph

The System Data tab displays a module-by-module breakdown of performance data, while the System Graph tab displays the same module-by-module breakdown of performance data but in chart form.

Figure 5.6. System Graph

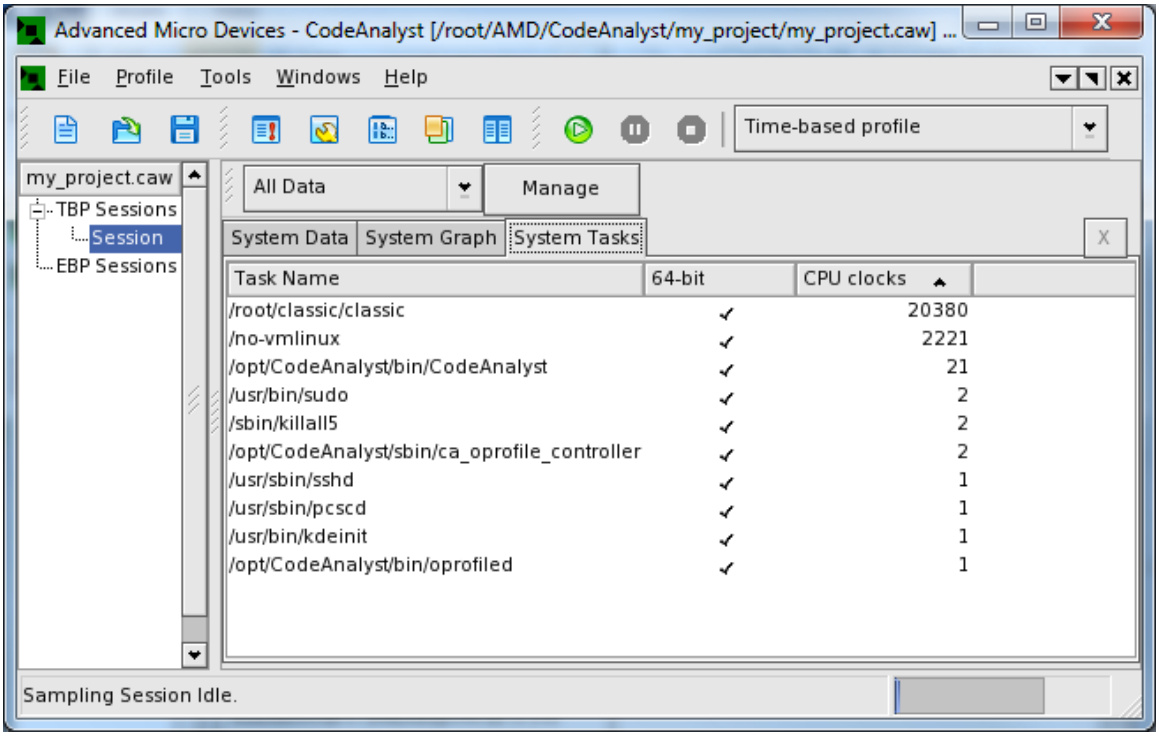


The configuration of both the System Data table and System Graph can be altered by selecting options in the Section 7.3, “View Management” window.

5.2.4. System Tasks

The System Tasks tab displays a task-by-task breakdown of performance data.

Figure 5.7. System Tasks



5.3. Collecting an Event-Based Profile

Event-based profiling (EBP) collects data based on how a program behaves on processors and memory. Information from this type of profile can be analyzed for developing and testing hypothesis about performance issues. AMD processors provide a wide range of hardware events that can be monitored and measured.

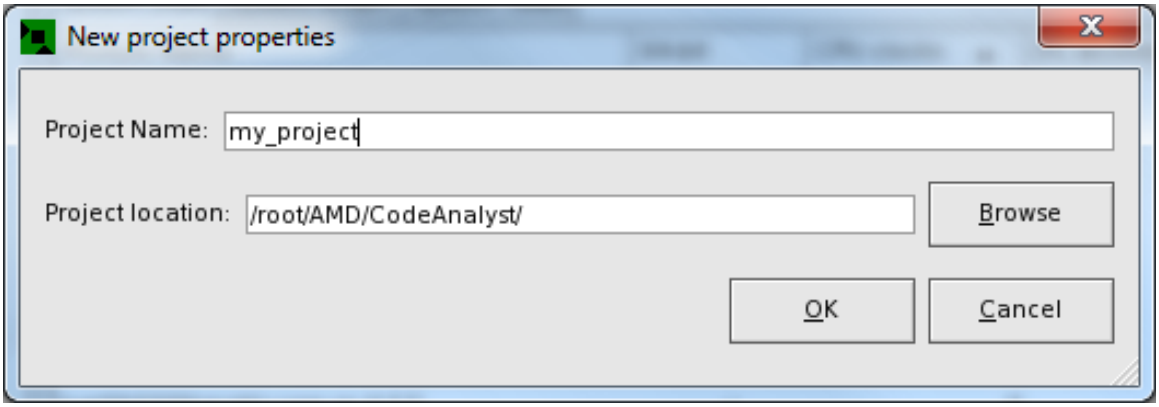
5.3.1. Collecting an Event-Based Profile

To collect an event-based profile :

1. Create new project or choose a previously opened project. Select an event-based profiling configuration like **"Assess performance"** from the drop-down profile configuration list in the toolbar.

2. If a new project is to be created, the **New project properties** dialog box opens. Assign a project name and location, or browse for an existing file.

Figure 5.8. New Project Properties

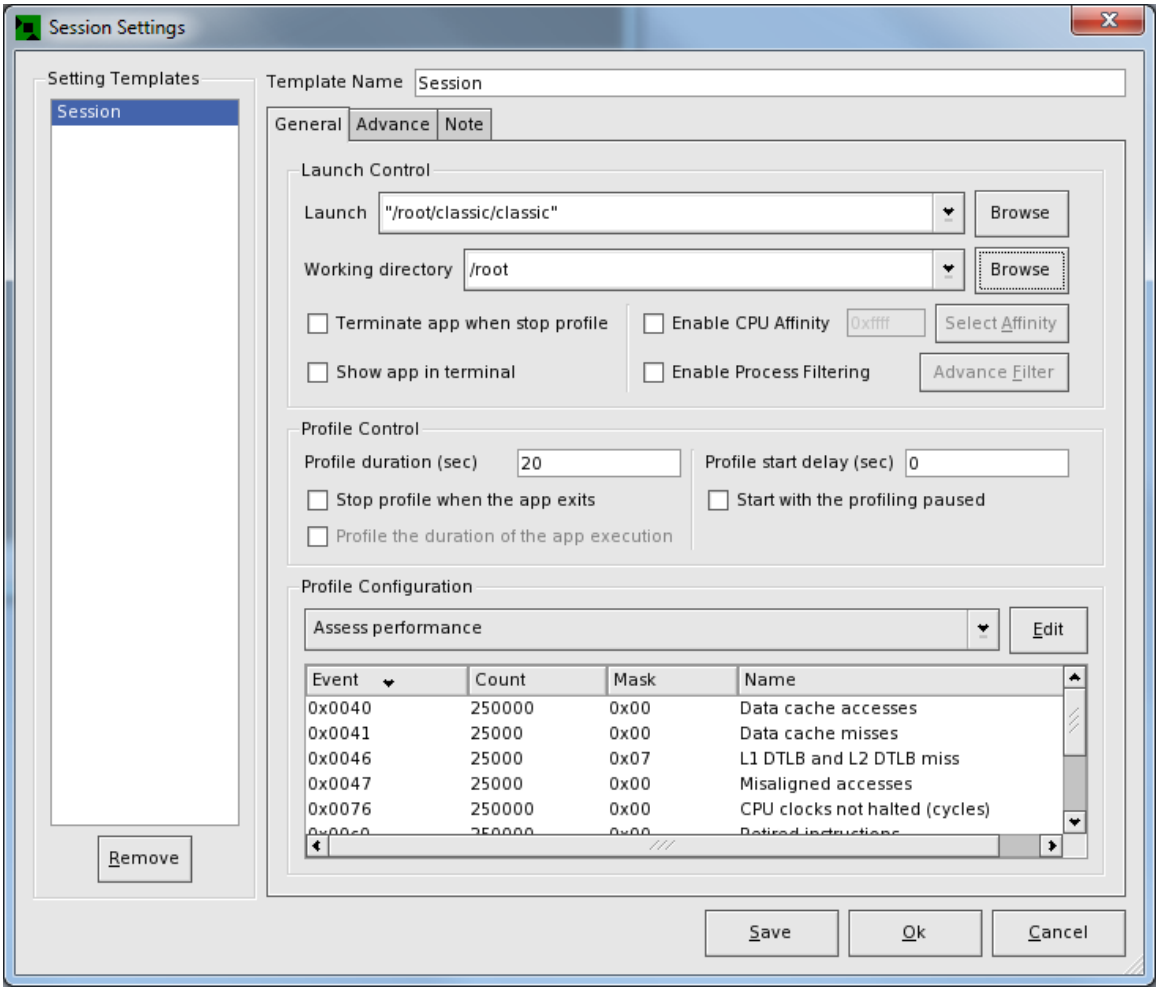


3. The **Session settings** dialog box opens. Assign a session name (optional), enter the path to the application program to be launched and set the working directory.

4. Under **Profile configuration:** select a predefined event-based profile configuration such as "Assess performance" and click OK.

5. Select other desired profiling options and click OK."

Figure 5.9. Session Settings

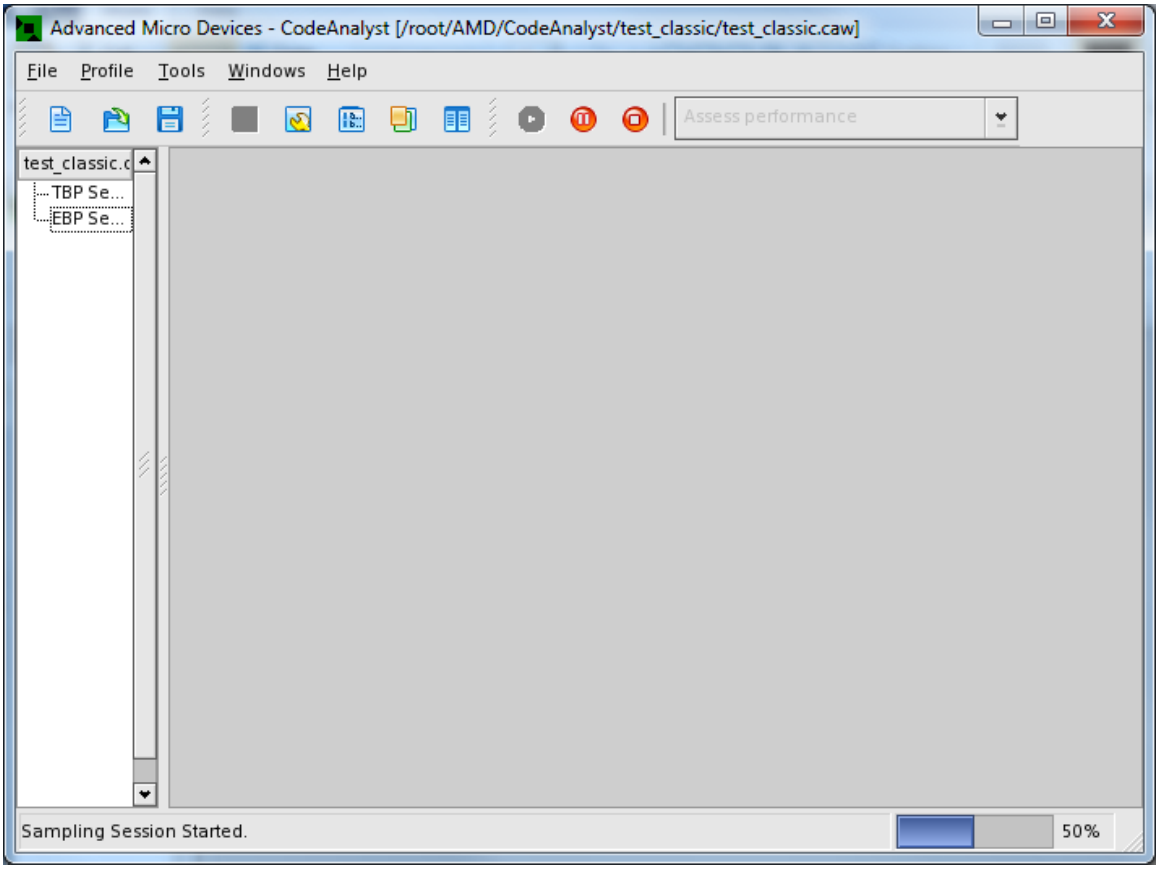


6. Click **Ok** to apply selections.

7. Click the Start icon  to launch the application and begin profiling.

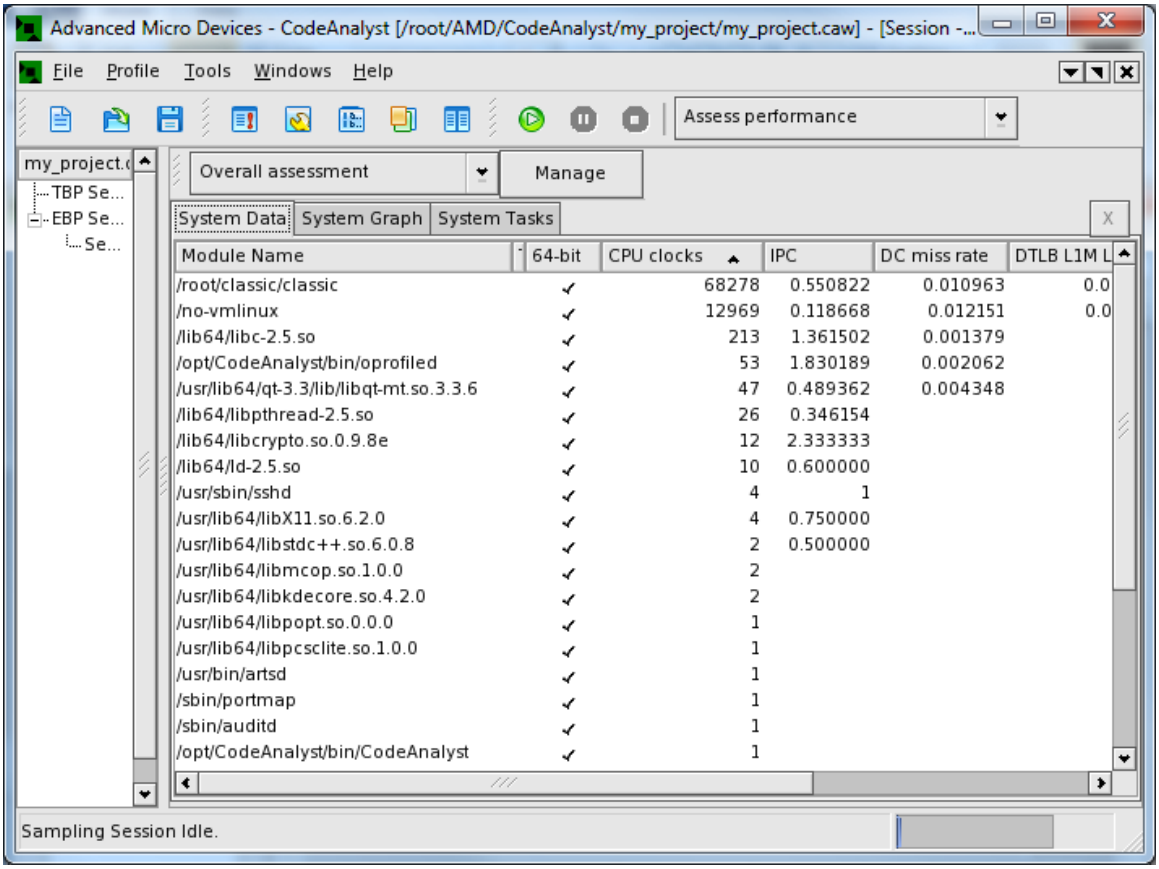
8. The task bar at the bottom of the screen displays "Sampling Session Started" and the percent completed. The **Pause** and **Stop** icons  become active.

Figure 5.10. Launch Application



9. When the sampling session is complete, the application under test terminates and the performance data is processed. The work space then displays a module-by-module breakdown of the results in the System Data table. Select the System Graph tab to see the results in graphical form. Select the System Tasks tab to see a task-by-task breakdown of the results. Double-click on a module or task to drill down into the data.

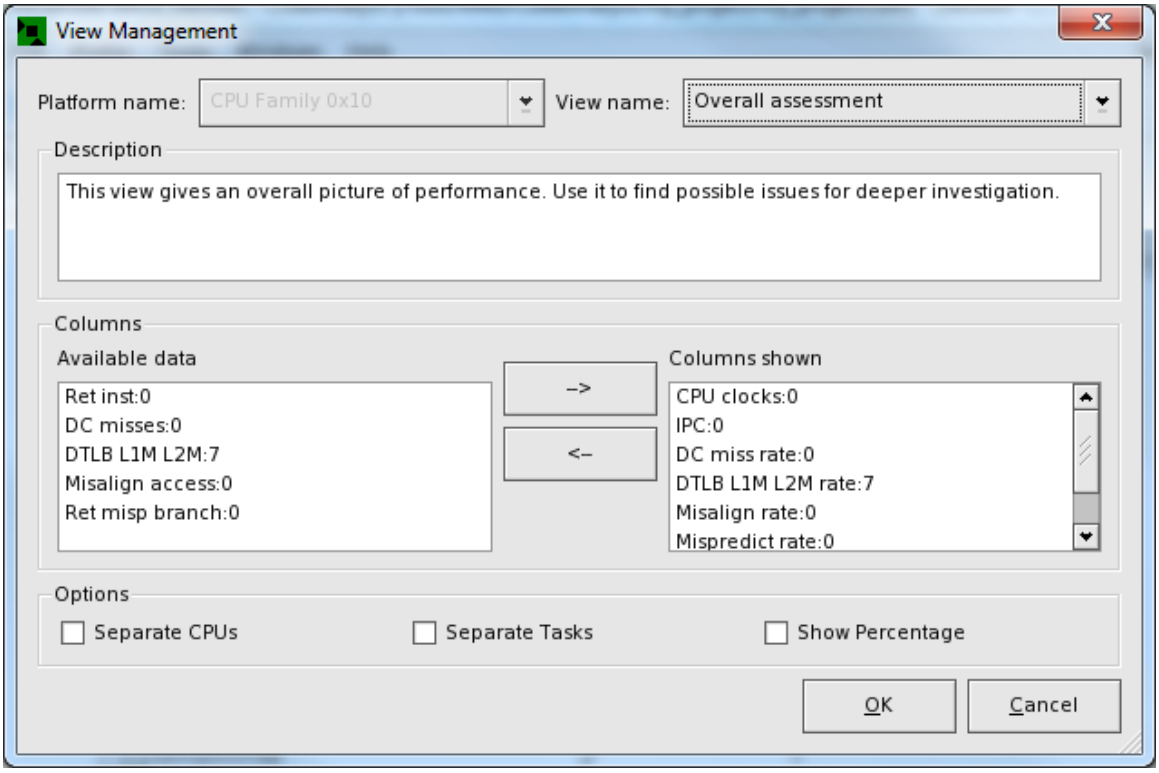
Figure 5.11. Performance Data Results



5.3.2. Changing the Current View of the Data

To change the type of data displayed in the current view, click **Manage**. The **View Management** dialog box opens. Refer to the Section 7.3, “View Management” section for details. The items listed in the Columns part of the View Management dialog box depend on the view configuration that is currently open for use.

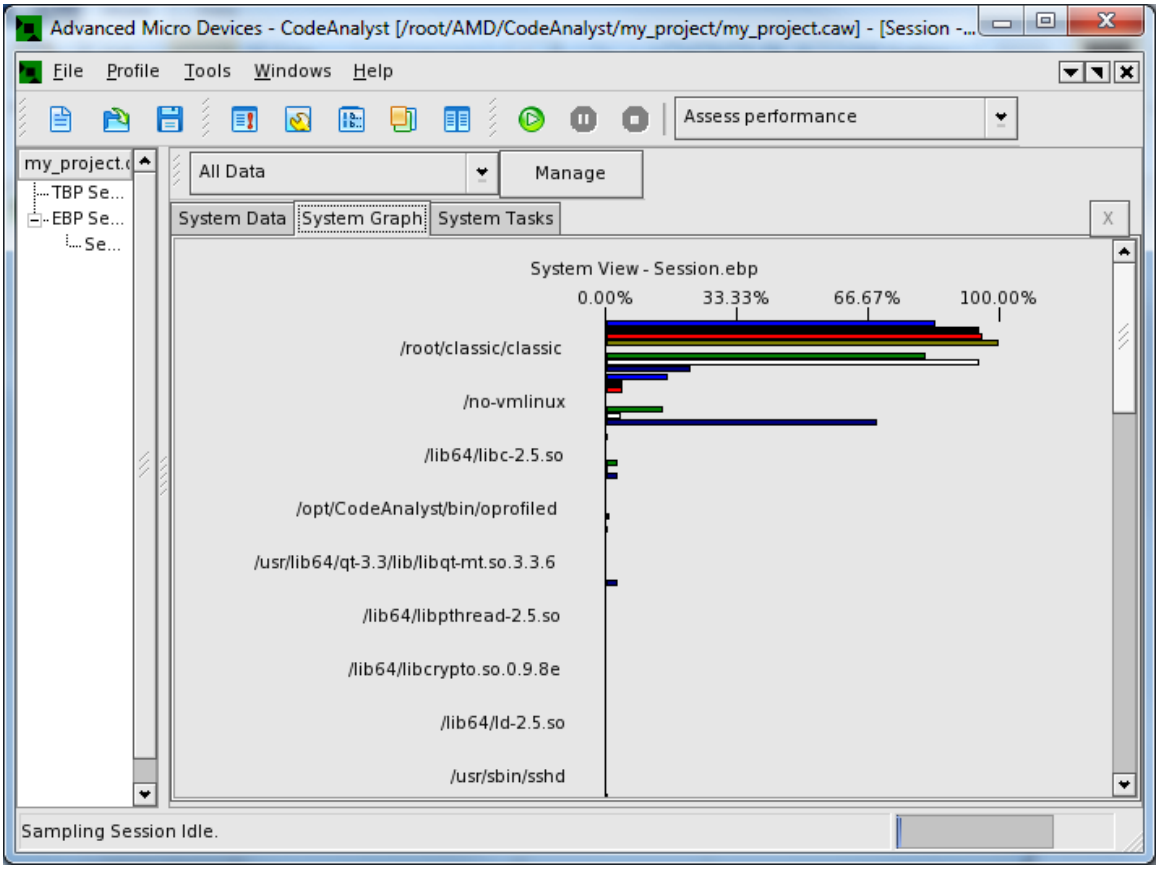
Figure 5.12. View Management



5.3.3. System Data and System Graph

The System Data tab displays a module-by-module breakdown of performance data, while the System Graph tab displays the same module-by-module breakdown of performance data but in chart form.

Figure 5.13. System Graph

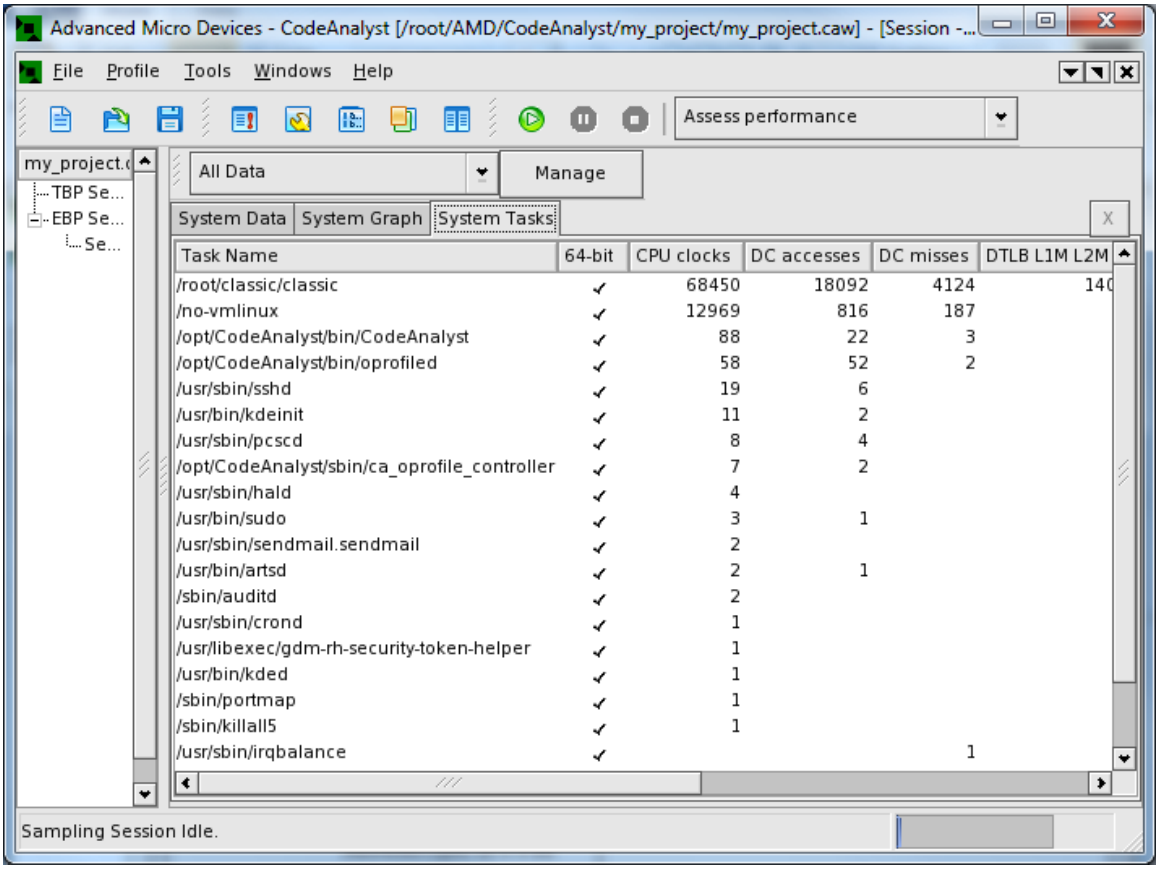


The configuration of both the System Data table and System Graph can be altered by selecting options in the Section 7.3, “View Management” window.

5.3.4. System Tasks

The System Tasks tab displays a task-by-task breakdown of performance data.

Figure 5.14. System Tasks



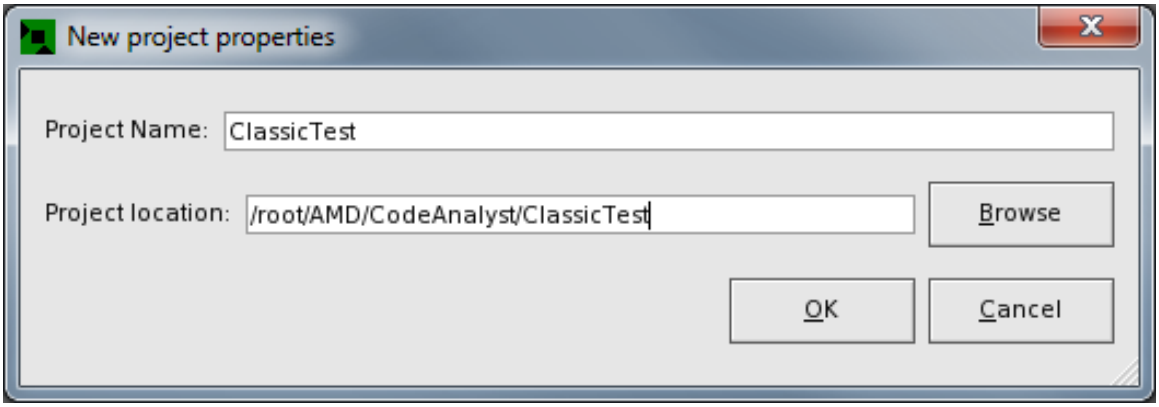
5.4. Collecting an Instruction-Based Sampling Profile

Instruction-based sampling (IBS) profile is similar to Event-based sampling (EBS) profile. It collects data based on how a program behaves on processors and memory. This section demonstrates how to configure CodeAnalyst to collect an instruction-based sampling profile.

5.4.1. Collecting an IBS Profile

1. Select **"Create new project"** or choose a previously opened project from the Welcome window. Select **"Instruction-Based Sampling"** from the second drop-down list in the toolbar. If a new project is to be created, the **"New project properties"** dialog box opens. Assign a project name and location or browse for an existing file.

Figure 5.15. New Project Properties

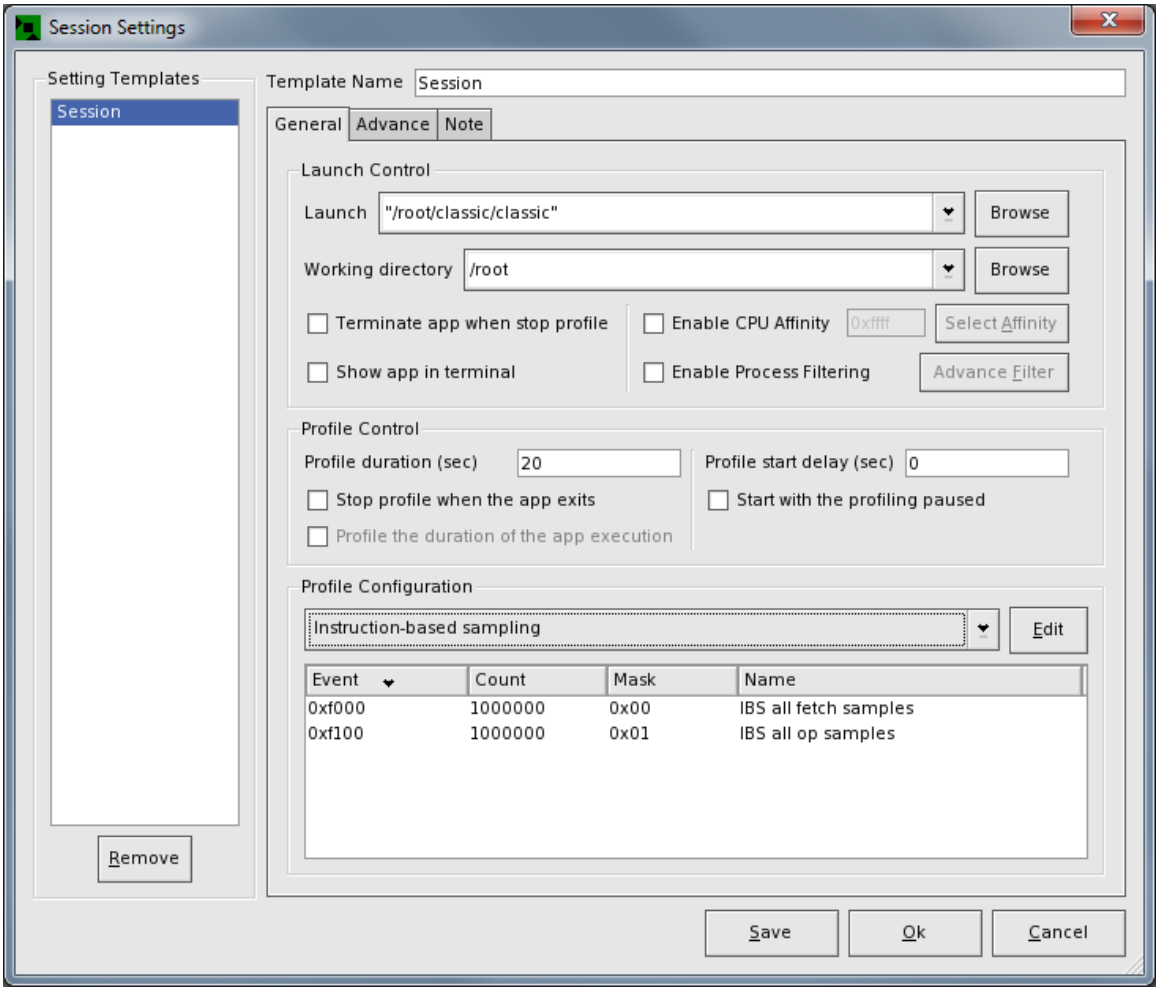


2. The **"Session settings"** dialog box opens.

   - In the **"Launch"** field, enter the path to the application program to be started, or browse to find the executable program.
   - The **"Working directory"** field fills automatically. You may also set the working directory to a different location by either entering the path directly or by browsing to it.
   - Under **"Profile configuration"** select the predefined profile configuration "Instruction-Based Sampling."

3. **Advanced step:** If editing the IBS profile configuration, click **"Edit"** to open the "Edit EBS/IBS Configuration" dialog box. (See Section 4.3, “Edit Event-based and Instruction-based Sampling Configuration” for more detail.

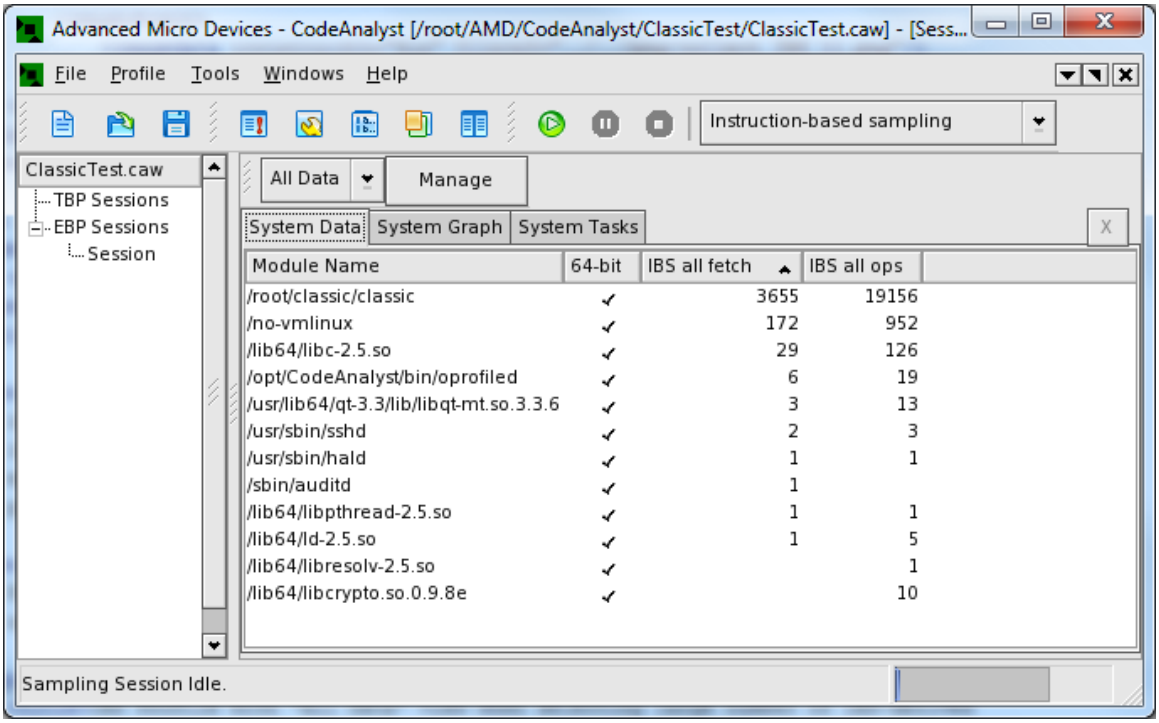
Figure 5.16. Session Settings



4. Click **"Ok"** to apply selections.

5. Click the **"Start"** icon  to launch the application and begin profiling. The task bar at the bottom of the screen displays "Sampling Session Started" and the percent completed. The **"Pause"** and **"Stop"** icons  become active. When the sampling session is complete, the application-under-test terminates and the performance data is processed. The work space then displays a module-by-module breakdown of the results in the System Data table. Select the System Graph tab to see the results in graphical form. Select the System Tasks tab to see a task-by-task breakdown of the results. Double-click on a module or process to drill down into the data.

Figure 5.17. Output from IBS Profile Session

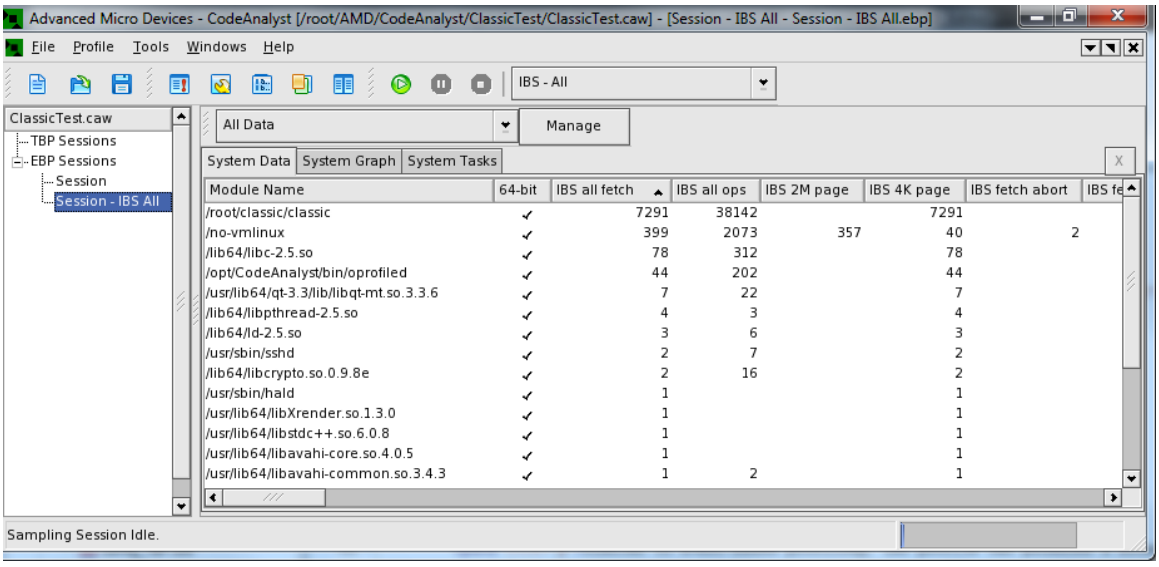


5.4.2. Changing the Current View of the Data

Similar to event-based profiling, IBS profile can produces a broad range of information about program behavior in a single run. IBS fetch samples provide information about instruction fetch while IBS op samples provide information about the execution of operations (ops) that are issued from AMD64 instructions. Several views provide a more focused look at different aspects of fetch and execution behavior.

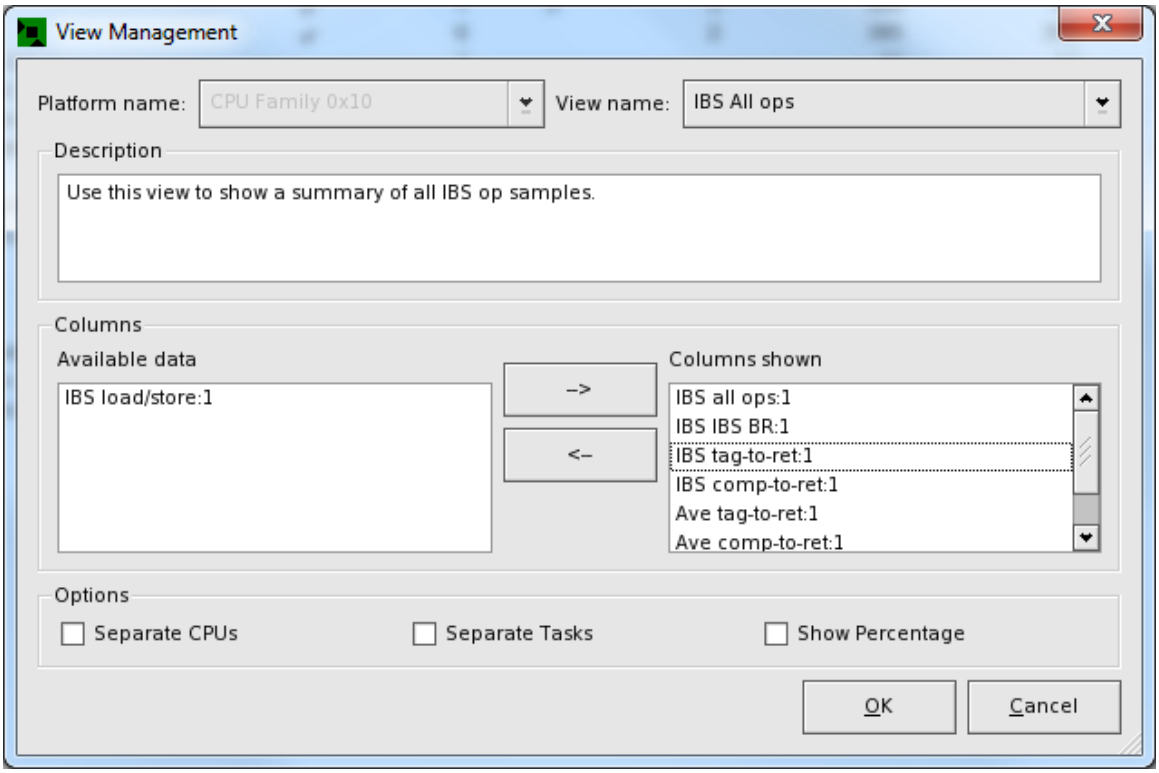
The "All Data" view displays sample counts for all IBS-derived events. Predefined views are provided to narrow down the displayed performance data to the most useful groups of IBS-derived events.

Figure 5.18. IBS Profile with "All Data" view when selecting large number of IBS-derived events



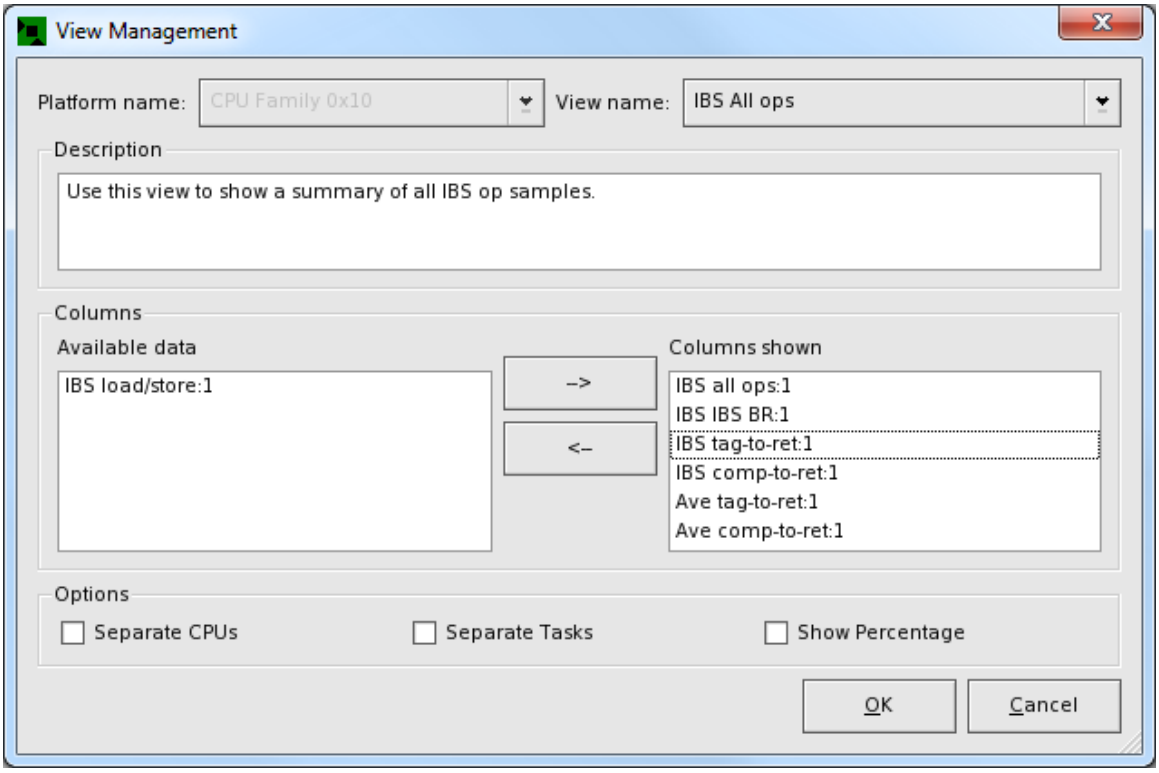
Use the drop-down list of views to select a different view of the IBS data. (The drop-down list is located next to the **"Manage"** button.) For instance, choose **"IBS fetch instruction cache"** to see a summary of IBS attempted fetches, completed fetches, instruction cache (IC) misses and the IC miss ratio, or choose **"IBS All ops"** from the drop-down list to see a summary of the number of all IBS op samples, IBS branch (BR) samples and IBS load/store samples that were collected.

Figure 5.19. IBS All Ops



To change the type of data displayed in the current view, click **Manage**. The **View Management** dialog box opens. Refer to the Section 7.3, “View Management” section for details. The items listed in the Columns part of the View Management dialog box depend on the view configuration that is currently open for use.

Figure 5.20. View Management

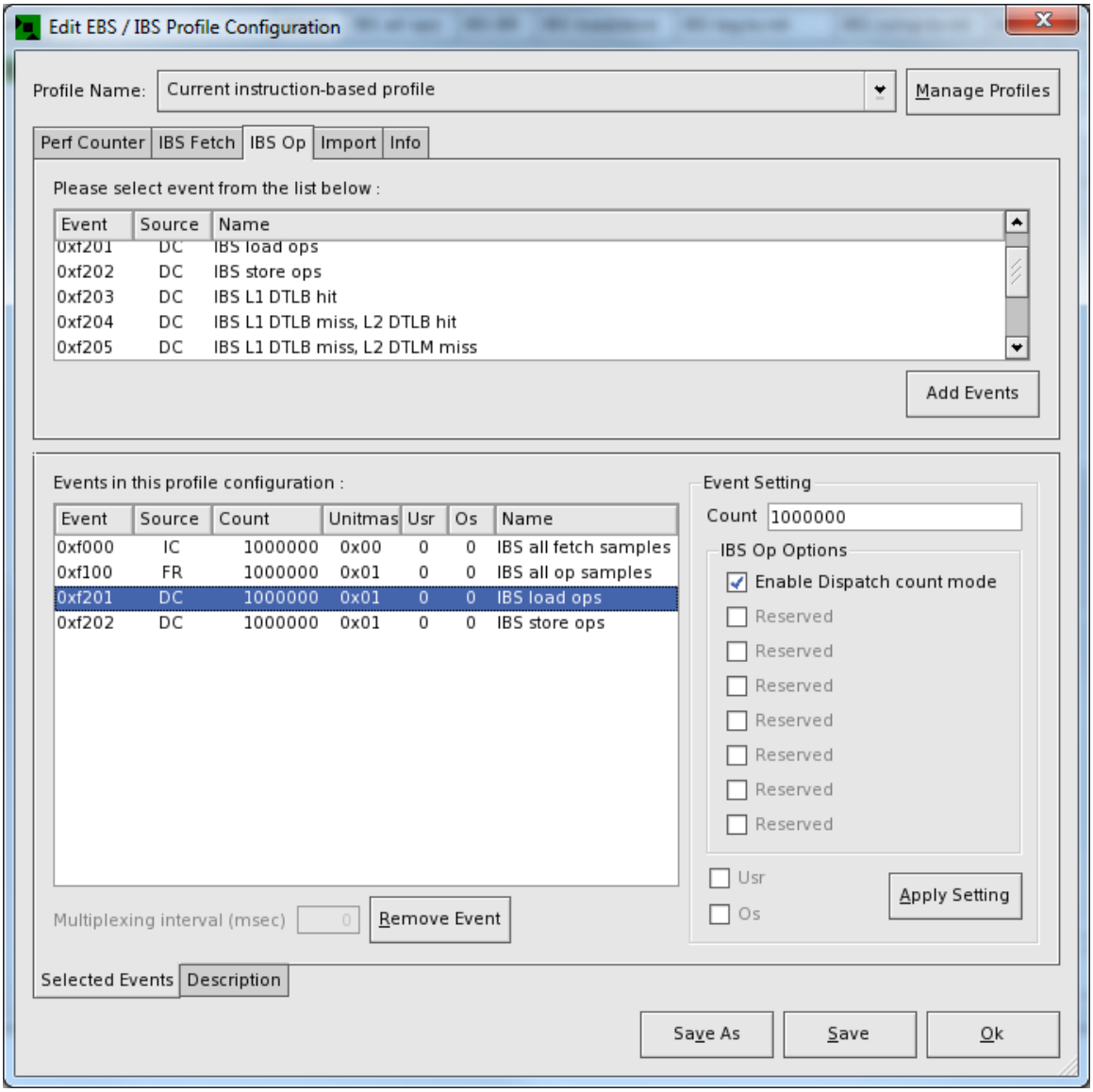


5.4.3. Changing How IBS Data is Collected

The predefined profile configuration named "Instruction-Based Sampling" collects both IBS fetch and op data. The way IBS data is collected can be changed by editing the "Current instruction-based profile" configuration.

1. Click the **"Session settings" button** in the toolbar or select **"Tools > Sessions Settings"** from the menu bar. Under **"Profile configuration"** select the profile configuration **"Current instruction-based profile"**.

Figure 5.21. Edit EBS/IBS configuration



2. In the "Edit EBS/IBS configuration" dialog, users can select IBS-Fetch/Op events from the available performance event lists, remove events from the selected list of events, or change the sampling periods and options of each type of event. Please see Section 4.3, “Edit Event-based and Instruction-based Sampling Configuration” for more detail.

3. Click **OK** to apply the changes.

Chapter 6. Data Collection Configuration

6.1. Data Collection Configuration

CodeAnalyst provides predefined profile (data collection) and view configurations to set up data collection and the presentation of results. The CodeAnalyst GUI provides the ability to change profile and view configurations.

CodeAnalyst stores these predefined configurations (and custom user configurations) in XML files. Expert users may choose to create or modify their own profile and view configurations by editing XML files directly. The two subsections below describe the XML format and tags for profile and view configuration files.

Section 6.2, “Profile Configuration File Format”

Section 7.5, “View Configuration File Format”

6.2. Profile Configuration File Format

A data collection configuration file describes how CodeAnalyst is to be configured for data collection. Through the CodeAnalyst GUI, the user chooses one of several data collection configurations. The GUI configures data collection according to the selected configuration.

The user may also modify a data collection configuration to fit their needs using the CodeAnalyst GUI. The modified data collection configuration can be saved to a file.

A data collection configuration file contains a single configuration. The file representation of a data collection configuration is in XML.

6.2.1. XML file format

6.2.1.1. Collection configuration

The following tags mark the beginning and end of configuration information within a data collection configuration file.

```
<dc_configuration>
  ...
</dc_configuration>
```

A collection configuration element contains one <tbp>, <ebp> or <sim> element. Each element describes a data collection configuration of the type indicated by its element name. Each such element describes how to configure CodeAnalyst to collect data.

6.2.1.2. TBP collection configuration

The <tbp> and </tbp> tags mark the beginning and end of a time-based profiling data collection configuration.

A <tbp> element has the following attributes:

name: Configuration name (string)

interval: Sampling interval given in milliseconds (float)

A TBP collection configuration element contains exactly one of the following elements: <tool_tip> and <description>. The <tool_tip> and <description> elements have a common form and are described below.

A time-based profiling configuration has the form:

```
<dc_configuration>
  <tbp name="..." interval="10.0">
    <tool_tip> ... </tool_tip>
    <description> ... </description>
  </tbp>
</dc_configuration>
```

6.2.1.3. EBP collection configuration

The <ebp> and </ebp> tags mark the beginning and end of a event-based profiling data collection configuration.

An <ebp> element has the following attributes:

name: Configuration name (string)

mux_period: Event multiplexing period in milliseconds (integer)

The mux_period attribute specified the event multiplexing period. If the mux_period is zero, event multiplexing is disabled. Event multiplexing should not be used when all events can be programmed onto the hardware counters in a profile run. The number hardware counters can be different on each platform on which measurements are to be taken. For example, AMD Athlon64™ and Opteron™ processors have four performance counters and thus, the number of <event> elements is limited to a maximum of four. Information from extra <event> elements may be discarded.

An EBP collection configuration element contains exactly one of the following elements: <tool_tip> and <description>.

An <event> element has the following attributes:

select: Event select value (integer)

mask: Unit mask value (integer)

os: Enables OS sampling (Boolean)

user: Enabled user-level sampling (Boolean)

count: Sampling period (integer)

edge_detect: Enable edge detect when counting events (Boolean)

host: Enable host mode event counting Boolean)

guest: Enable guest mode event counting (Boolean)

The values must be validated against the events and specific capabilities supported by the measurement platform.

An event-based profiling data collection configuration has the form:

```
<dc_configuration>
  <ebp name ="..."  mux_period="10">
    <tool_tip> ... </tool_tip>
    <description> ... </description>
      <event select="0x00" mask="0x00" os="T" user="T"
         host="F" guest="F" edge_detect="F" count="50000"></event>
      ...
  </ebp>
</dc_configuration>
```

6.2.1.4. Tool tip

The tags <tool_tip> and </tool_tip> mark the beginning and end of a short "tool tip" description of a configuration. The text between the tags is the tool tip description.

```
<tool_tip> ... </tool_tip>
```

A tool tip is usually only a few key words with no line breaks.

6.2.1.5. Description

The tags <description> and </description> mark the beginning and end of a short description of a configuration. The text between the tags is the description.

```
<description> ... </description>
```

A description is usually only a few sentences long. It may contain line breaks. Line breaks and extra space will be simplified. Line breaks will be replaced by spaces and runs of space characters will be replaced by single space characters.

6.2.1.6. Attributes

The value of a numeric attribute is returned as a string that can be converted to internal integer and floating point representation. The actual values may be further constrained for the attribute's purpose, e.g., the number of simulated instructions must be greater than 0. The default value for a numeric attribute is zero.

The value of a Boolean attribute is returned as one of the strings "T" or "F", which denote the truth values true and false, respectively. The default value of a Boolean attribute is "F".

6.2.2. Examples of XML Files

6.2.2.1. TBP example

Here is an example of a data collection configuration for time-based profiling.

```
<dc_configuration>
  <tbp name="Time-based profile" interval="10.0">
    <tool_tip> Measure time in program regions </tool_tip>
    <description>
      Monitor the program and produce a profile showing where the program
      spends its time. Use this configuration to identify hot-spots.
    </description>
  </tbp>
```

```
</dc_configuration>
```

6.2.2.2. EBP example

Here is an example of a data collection configuration for event-based profiling.

```
<dc_configuration>
  <ebp name ="Quick tune"  mux_period="100000">

      <!-- CPU clocks unhalted -->
      <event select="0x76" mask="0x00" os="T" user="T" count="50000"></event>

      <!-- Retired instructions -->
      <event select="0xC0" mask="0x00" os="T" user="T" count="50000"></event>

      <!-- In order to detect branch mispredictions -->
      <event select="0xC2" mask="0x00" os="T" user="T" count="5000"></event>

      <!-- Mispredicted branches -->
      <event select="0xC3" mask="0x00" os="T" user="T" count="5000"></event>

      <!-- Data cache accesses -->
      <event select="0x40" mask="0x00" os="T" user="T" count="5000"></event>

      <!-- Data cache misses -->
      <event select="0x41" mask="0x00" os="T" user="T" count="5000"></event>

      <!-- L1 DTLB and L2 DTLB Miss -->
      <event select="0x46" mask="0x00" os="T" user="T" count="5000"></event>

      <!-- Misaligned accesses -->
      <event select="0x47" mask="0x00" os="T" user="T" count="5000"></event>

    <tool_tip> Perform quick accessment </tool_tip>

    <description>
      Collect data needed to quickly identify possible performance issues.
      Good candidates for further investigation are frequently executed,
      time-consuming parts of the program. Use analysis configurations for
      follow-up investigations.
    </description>
  </ebp>
</dc_configuration>
```

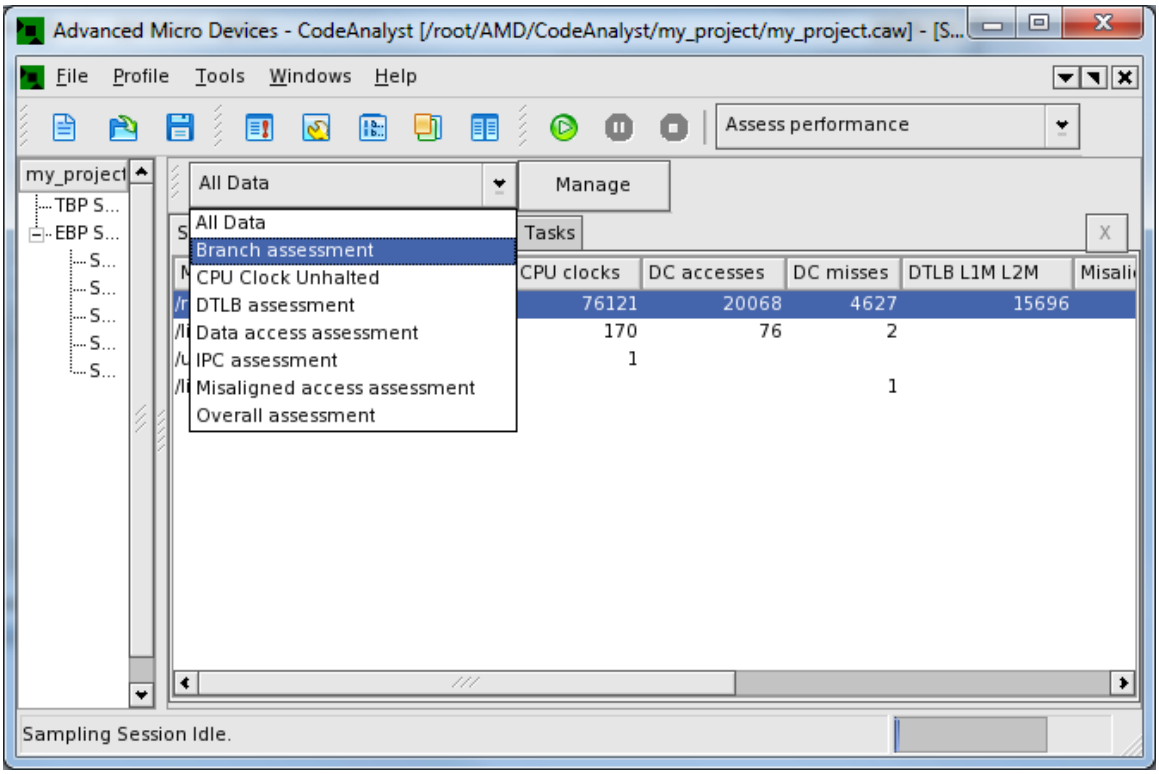
Chapter 7. View Configuration

7.1. Viewing Results

AMD CodeAnalyst can collect many different kinds of performance data in a single measurement session. This is especially true of event-based profiling and Instruction-Based Sampling where a large number of events may be collected in a single performance experiment. The CodeAnalyst **view** feature organizes performance data into groups of logically related kinds of data in order to aid interpretation and analysis.

Select a view from the drop-down list of views to change to a different view.

Figure 7.1. List of Views



The **All Data** view is displayed by default after data collection. The data is displayed in as many columns as there are distinct kinds of sample data.

The list of available views is determined by the kind of performance data that was collected. A view is only offered if the right kind of performance data is available to produce the view. CodeAnalyst provides several predefined views. See Section 7.4, “Predefined Views” for more details.

The choice of performance data and the aggregation/separation of data by CPU (core), process and thread is controlled through Section 7.3, “View Management”. View Management aids interpretation by filtering out unneeded performance data and by breaking out (or summarizing) performance data across CPUs, processes and threads.

The CodeAnalyst GUI does not provide a way to directly create and save new views. Advanced users may choose to create their own views and should see Section 7.2, “View Configurations” for more information. Expertise with XML is required.

7.2. View Configurations

Information that defines a view is stored in a **view configuration**. View configurations -- like profile configurations -- are stored as .XML format files. This approach allows for:

- Sharing view configurations between users.
- Saving customized view configurations.
- Creating and distributing new view configurations more easily.

Predefined views are installed as .XML files and cannot be permanently changed. Any changes made in a session only appear while the session is open. Advanced customization is possible by making a copy of the .XML file containing the view configuration and then modifying the XML representation itself.

The view configuration files that are installed with CodeAnalyst provide a good starting point for creation of new views. Predefined view configuration files are in the **/opt/CodeAnalyst/share/codeanalyst/Configs/ViewConfig** directory. In this directory, CodeAnalyst stores common view configuration files. Platform specific configuration files are stored in the subdirectories with the corresponding AMD processor family name.

NOTE : Since the unit mask for some performance monitoring events vary on different AMD processor families, view configurations that use these events are listed underneath platform specific directory.

See Section 7.5, “View Configuration File Format” for more information about the XML representation of a view.

7.3. View Management

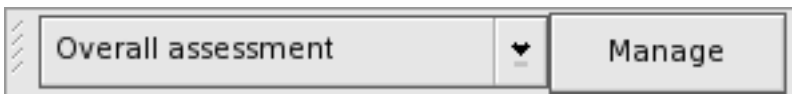
The content and format of a view can be modified using **view management**. Use view management to change the kinds of performance data shown in a view, to aggregate or separate data by core (CPU), to aggregate or separate data by task, to aggregate or separate data by thread, and to control display of a percentage column.

Two types of View Management dialog are available:

- **Global :** Manage a collection of session dialogs that are currently using the same view. The changes made to a view configuration within the dialog will propagate to all opening sessions that are currently displaying the view. Global View Management dialog also allows user to manage platform specific view configurations. Changes made in this dialog are saved to the view configuration file. The user can reset the pre-defined view configuration to the original value in this dialog.
- **Local :** Manage the view of an individual session. The changes made to a view configuration within the dialog are applied only to an individual session, and are not saved to the view configuration file.

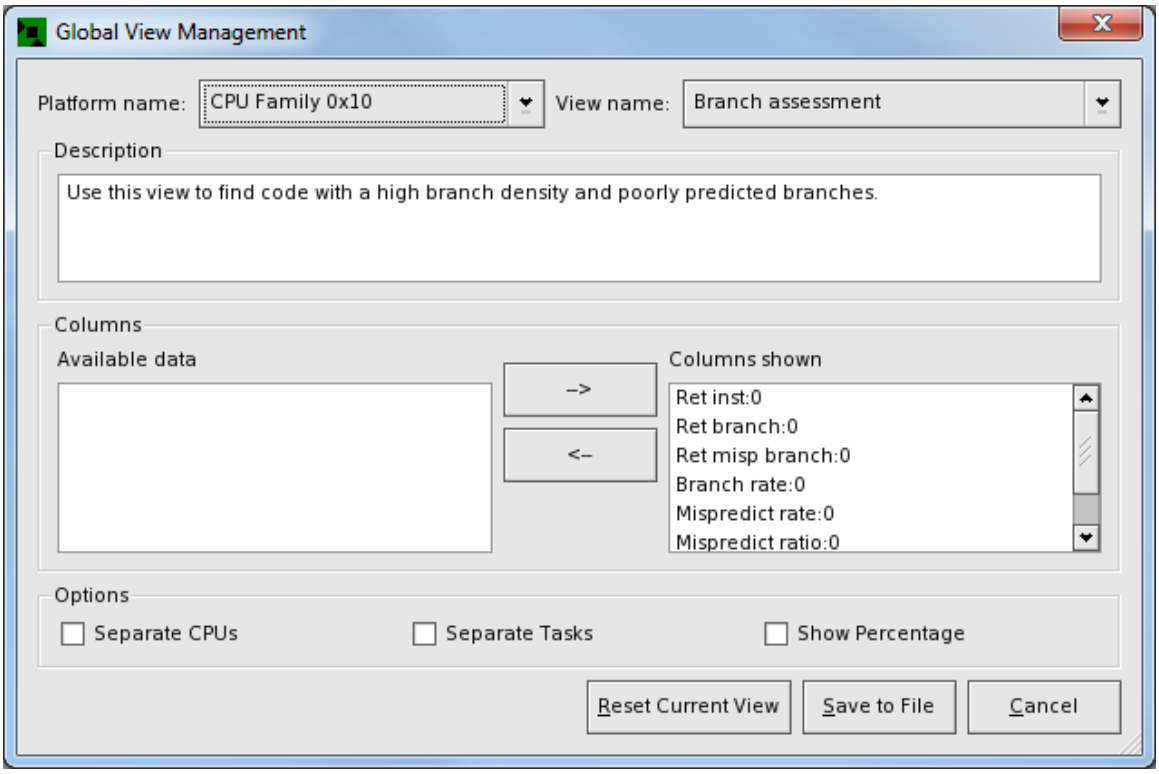
View management can be initiated in any of three ways:

- Click the view management icon  in the toolbar. **(Global)**
- Select **Tools > View Management** from the menu bar. **(Global)**
- Click the **Manage** button next to the drop-down list of views. **(Local)**



7.3.1. View Management Dialog

Figure 7.2. Global View Management Dialog



7.3.1.1. Platform Name

The Platform Name field shows the name of the selected AMD processor family. Processor families are selected from a drop-down list, which is enabled only in the Global View dialog. In Local View dialog, the processor type is determined by the type of processor the profile data of the session was collected on.

7.3.1.2. View Name

The View Name field shows the name of the selected view configuration. View configurations are selected from a drop-down list.

7.3.1.3. Description

The description changes according to the selection in View name.

7.3.1.4. Columns

CodeAnalyst can only display items listed in the Available data list.

- **Available data:** The items in the Available data list are preselected according to the selected view configuration. Each item is available for inclusion in a view as a column.
- **Columns shown:** Selects which columns are shown and can be changed by using the directional arrows to move an item from the list.

The selected platform name in **Platform Name** drop-down menu list determines the types of AMD processor family, which in turn determines the available views shown in **View name** drop-down menu list. Availability of views are determined by the available performance counters on a particular AMD processor family. The selected view determines the choices displayed under **Available data** and under

Columns shown. The performance data to be shown in the selected view can be changed by moving items between the **Available data** list and the **Columns shown** list.

To add an item to the **Columns shown** list and make it visible in the view:

1. Select the item in the **Available data** list.

2. Click the right arrow -> button.

To remove item from the **Columns shown** list and remove it from the view:

1. Select the item in the **Columns shown** list.

2. Click the left arrow <- button.

NOTE : Changes made to view configurations are persistent.

7.3.1.5. Separate CPUs

When selected, event data is shown for each processor in separate columns in the System Data table and as separate bars in the System Data graph. For example, on a dual-core processor, timer-based samples display in two columns with the headings:

- Time-based samples - C0
- Time-based samples - C1

7.3.1.6. Separate Tasks

For Separate Tasks, a separate row is displayed in the System Data table for each unique use of a module by a task. The module name is extended by the task name. A new bar is added in the System Data graph for each unique use of a module by a task.

7.3.1.7. Show Percentage

Enabling Show Percentage displays the percentage column in the view.

7.4. Predefined Views

AMD CodeAnalyst provides many predefined views. These predefined views cover the most common kinds of analysis and directly support the predefined profile configurations that control collection of performance data. Profile (data collection) configurations choose which performance data are collected during a session. Coordinated, predefined view configurations provide purpose-specific views of this performance data. A view may also include computed performance measurements (rates and ratios of events, for example) that are calculated from the raw performance data.

View availability is determined by the availability of the performance data required for generating the view. Any view that can be displayed from available performance data can be selected for display. For example, the misaligned accesses view is offered when data is collected using either the basic assessment and the data access profile configurations because both of these predefined profile configurations collect the events needed to display the misaligned accesses view.

The following table summarizes the predefined view configurations associated with each of the predefined profile configurations.

Profile Configuration	**View Configuration**
Assess performance	• Overall assessment • IPC assessment • Branch assessment

	• Data access assessment • DTLB assessment • Misaligned access assessment
Investigate branching	• Branch assessment • Near return report • Taken branch report
Investigate data access	• Data access assessment • Data access report • DTLB assessment • DTLB report • Misaligned access assessment
Investigate instruction access	• Instruction cache report • ITLB report
Investigate L2 cache access	• L2 access report
Instruction-based sampling	• IBS fetch overall • IBS fetch instruction cache • IBS fetch instruction TLB • IBS fetch page translations • IBS All ops • IBS BR branch • IBS BR return • IBS MEM all load/store • IBS MEM data TLB • IBS MEM data cache • IBS MEM forwarding and band conflicts • IBS MEM locked ops and access by type • IBS MEM translations by page size • IBS NB cache state • IBS NB local/remote • IBS NB request breakdown

7.5. View Configuration File Format

View configurations control the content and formatting of CodeAnalyst views. The CodeAnalyst GUI offers predefined view configurations which can be selected by a user. View Configurations are stored

in .XML formatted files. A new customized view can be created by writing an .XML file for the view. Only the most advanced CodeAnalyst users should ever need to create views at the XML level. This section describes the format and semantics of a view configuration in XML format.

7.5.1. XML File Format

A view configuration file contains a single, non-empty <view_configuration> element. This element, in turn, contains a single, non-empty <view> element. This format allows the possibility of representing multiple views within a single configuration file. The current implementation assumes that only one <view> will be defined in a configuration file.

The <view> element describes a view and has the following attributes:

- **name:** Displayable symbolic name given to the view (string)
- **separate_cpus:** Controls aggregation/separation by CPU (Boolean)
- **separate_processes:** Controls aggregation/separation by process (Boolean)
- **separate_threads:** Controls aggregation/separation by thread (Boolean)
- **default_view:** Identifies this view as a candidate default view (Boolean)
- **show_percentage:** Toggle the format of the profile data shown between raw and percentage

Boolean attributes are restricted to the string values "T" and "F".

The view's name will appear in a list of available views and should suggest the purpose of the view to the user. The separation/aggregation attributes are optional and default to false. CodeAnalyst shows a view after data collection, a so-called "default view." The default_view attribute is a hint to CodeAnalyst that the view could (or should) be selected as a default view. Please use this attribute sparingly. The default_view attribute is optional and it's default value is false.

7.5.1.1. <view>

A <view> element contains one each of the following elements:

<data>	Describes the event data needed to show the view
<output>	Describes how the event data will be shown
<tool_tip>	Tool tip to be shown for the view
<description>	Short description to be shown for the view

Combining these elements, a view configuration file has the overall structure shown here:

```
<view_configuration>
   <view name="View name"
         separate_cpus="T" separate_processes="F" separate_threads="F">
      <data>
         <!-- List of event data needed -->
      </data>
      <output>
         <!-- List of "columns" to be shown -->
      </output>
      <tool_tip>
         Tool tip text
      </tool_tip>
      <description>
```

```
            Short descriptive text
          </description>
        </view>
    </view_configuration>
```

The <data> element specifies the event data that is needed to produce the view. CodeAnalyst uses this information to determine if a view is producable from a data set and will not offer the view to the user if it cannot be produced from available event data in the session. The <data> element must appear before the <output> element since the <output> element makes use of identifiers which are defined in the <data> element.

7.5.1.2. <data>

The <data> element contains one or more <event> elements. An <event> element:

- Indicates that event data of that particular type is needed to produce the view, and
- Defines a mnemonic, symbolic identifier which is used to refer to that event data in the <output> section of the XML file.

The <data> element may contain <event> elements for event data which is not used in the <output> section. Such <event> elements can be used to further control the association of a view with certain data collection configurations. For example, a view can be associated with a specific data collection configuration by listing exactly the events measured by the data collection configuration.

7.5.1.2.1. A <event>

An <event> element has three attributes:

id: Symbolic name given to the event data

select: Event selection value (hexadecimal integer)

mask: Event selection unit mask (hexadecimal integer)

The select and mask attributes take values as specified by the definition of performance events in the BIOS and Kernel Developers Guide (BKDG.) CodeAnalyst may choose to ignore the mask attribute when selecting event data.

With respect to future expansion and enhancement, new elements like the <event> element may be defined to accomodate new hardware performance measurement features.

7.5.1.3. A <output>

The <output> element specifies how event data will be shown. The <output> element contains one or more <column> elements. Each <column> element specifies a column of event data in a table. (The notion of column may be extended to a dimension in a graphical chart.) A <column> element has the following attributes:

title: Title or legend to be displayed along with the data (string)

sort: Controls sorting of data in the column (string)

visible: Controls column visibility (Boolean)

The title attribute is a descriptive string that is used to label the data in a table or graph. The sort attribute specifies whether the data should be sorted or not. The sort attribute has only three permitted values: "ascending", "descending" and "none". At most one column should have a sort attribute of "ascending" or "descending"; all other columns should be "none". The visible attribute is optional and has the default value true. A column may be hidden by setting the visible attribute to false. Generally, you should not need to define a hidden column.

7.5.1.3.1. <column>

A <column> element is non-empty and contains exactly one of the following empty elements:

<value>	Show the selected event data as a simple value
<sum>	Show the sum of two selected event data
<difference>	Show the difference of two selected event data
<product>	Show the product of two selected event data
<ratio>	Show the ratio of two selected event data

These elements select and combine the event data to be shown in the column. A <value> element has a single attribute:

id: Data identifier to select event data

The <sum>, <difference>, <product> and <ratio> elements take two attributes:

left: Data identifier to select data for the left operand

right: Data identifier to select data for the right operand

In the case of <ratio>, for example, the left identifier specifies event data for the numerator and the right identifier specifies event data for the denominator.

Note: Simple <value> elements using an event must appear before any <sum>, <ratio>, ... elements in the <output> section that use the same event. This ordering is required by the CodeAnalyst user interface.

With respect to future enhancement, additional elements like <sum>, etc. can be defined to combine event data. One such addition may be a <formula> element that defines a formula to be evaluated using event data. The formula may use identifiers to refer to event data.

7.5.1.4. A <tool_tip>

A <tool_tip> element is a non-empty XML element that contains the text of a tool tip to be displayed for the view. A <description> element is a non-empty XML element that contains the text of a short description to be displayed for the view. Leading and trailing space is trimmed from tool tip and description text. Spaces are compressed and new line characters are replaced by a single space.

7.5.2. Example XML File

Here is an example view configuration file. Note that several optional attributes are not used in the example and they take on the appropriate default values.

```
<!--
        Show instructions per cycle
        View configuration
        Date:    17 May 2006
        Version: 0.1
 -->

<view_configuration>

  <view name="IPC"
        separate_cpus="F"
        separate_processes="F"
```

```
            separate_threads="F"
            >

          <data>
            <event id="CPU_clocks" select="76" mask="00" />
            <event id="Ret_instructions" select="c0" mask="00" />
          </data>

          <output>
            <column title="Instructions" sort="none">
              <value id="Ret_instructions" />
            </column>
            <column title="Cycles" sort="none">
              <value id="CPU_clocks" />
            </column>
            <column title="IPC" sort="descending">
              <ratio left="Ret_instructions" right="CPU_clocks" />
            </column>
            <column title="CPI" sort="none">
              <ratio left="CPU_clocks" right="Ret_instructions" />
            </column>
          </output>

          <tool_tip>
            Show instructions per cycle
          </tool_tip>

          <description>
            Use this view to find hot-spots with low instruction level paralellism.
          </description>

        </view>

      </view_configuration>
```

Chapter 8. Tutorial

8.1. AMD CodeAnalyst Tutorial

This tutorial demonstrates how to use AMD CodeAnalyst to analyze the performance of an application program. The tutorial provides step-by-step directions for using AMD CodeAnalyst. We recommend reading the tutorial sections in the order listed below.

- Section 8.2, “Tutorial - Prepare Application”
- Section 8.3, “Tutorial - Creating a CodeAnalyst Project”
- Section 8.4, “Tutorial - Analysis with Time-Based Sampling Profile”
- Section 8.5, “Tutorial - Analysis with Event-Based Sampling Profile”
- Section 8.6, “Tutorial - Analysis with Instruction-Based Sampling Profile”
- Section 8.7, “Tutorial - Profiling a Java Application”

8.1.1. Related Topics

For quick reference to options available in the CodeAnalyst workspace, See Section 2.2, “Exploring the Workspace and GUI”

8.2. Tutorial - Prepare Application

This tutorial uses the example program that is distributed with AMD CodeAnalyst. Source code for the example program is installed with CodeAnalyst. To find the source code, locate the directory into which CodeAnalyst was installed and then find the `samples/classic` directory.

The example program, **classic**, implements the straightforward "textbook" algorithm for matrix multiplication. Matrix multiplication is performed in the function **multiply_matrices()**. This function provides an opportunity for optimization. The classic implementation takes long, non-unit strides through one of the operand matrices. These long strides cause frequent data translation lookaside buffer (DTLB) misses that penalize execution.

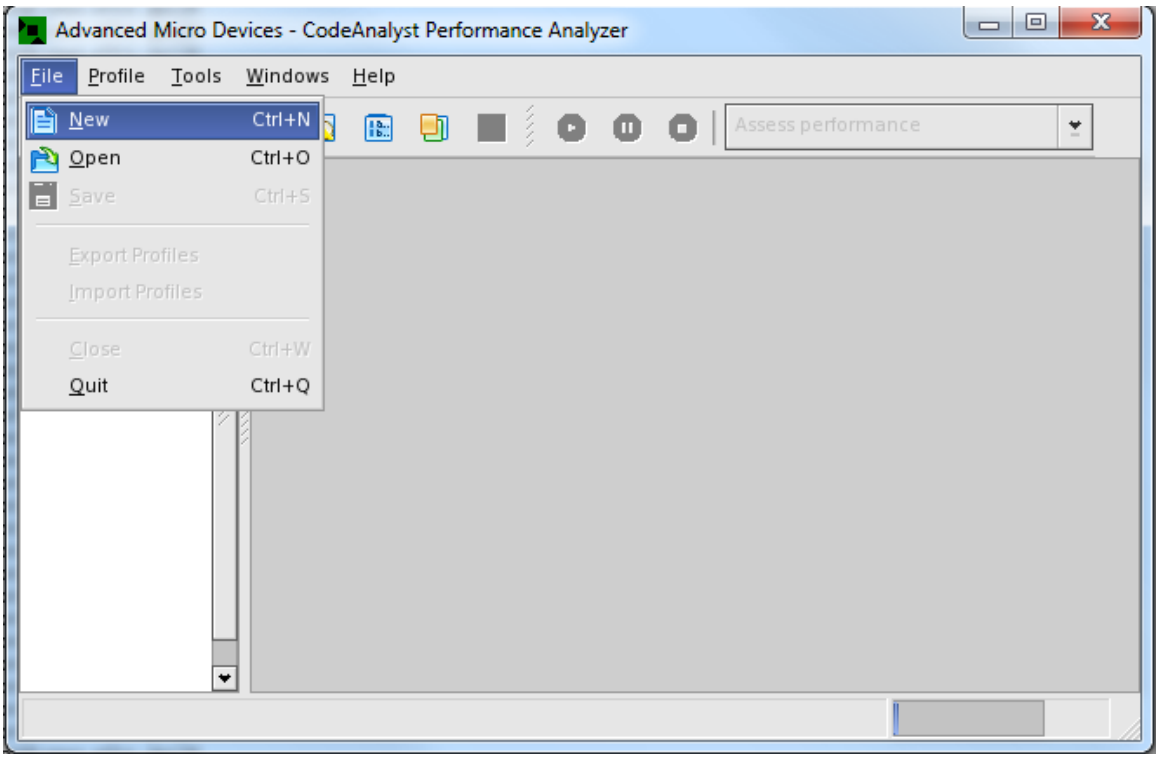
In order to prepare the example program from the tutorial,

1. Go to directory `/path_to_CodeAnalyst_source_directory/samples/classic/`.
2. At command prompt, run **g++ -g -o classic classic.cpp**. This should compile classic.cpp into classic binary with debugging information.

8.3. Tutorial - Creating a CodeAnalyst Project

This section demonstrates how to create a new CodeAnalyst project. All work takes place in the context of a CodeAnalyst project. You must either create a new project or open an existing project in order to collect and analyze performance data. Performance data is organized and saved as a “Session.” A project may contain multiple sessions. Sessions are saved with a project.

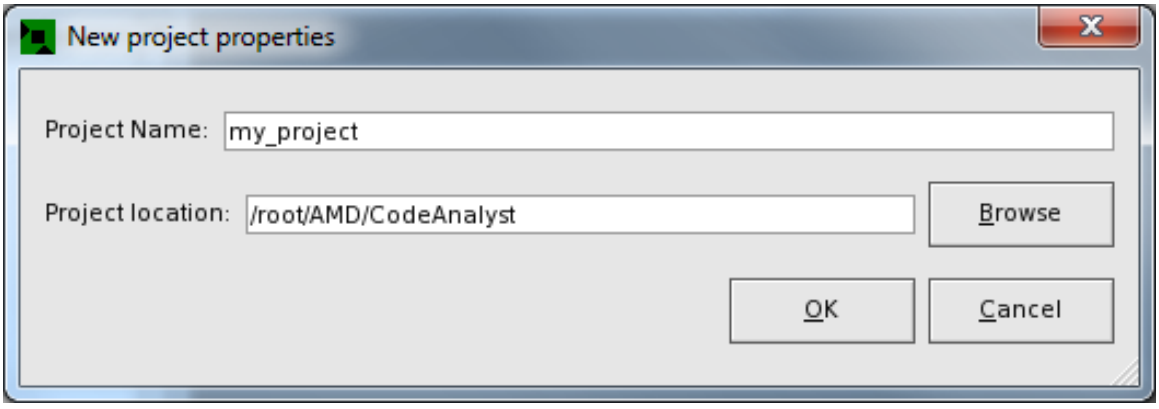
1. Launch CodeAnalyst. The CodeAnalyst window displays.
2. To create new project, click the **Newproject** icon in the toolbar or select **File > New** from the File menu. A dialog box appears asking for the project name and the project location. You must name the project and you must specify its location.



3. Enter the name of the new project into the **Project Name** field.

4. Enter the path to the directory that will hold the new project into the **Project location** field. You may also browse to the project location by clicking the **Browse** button next to the project location field.

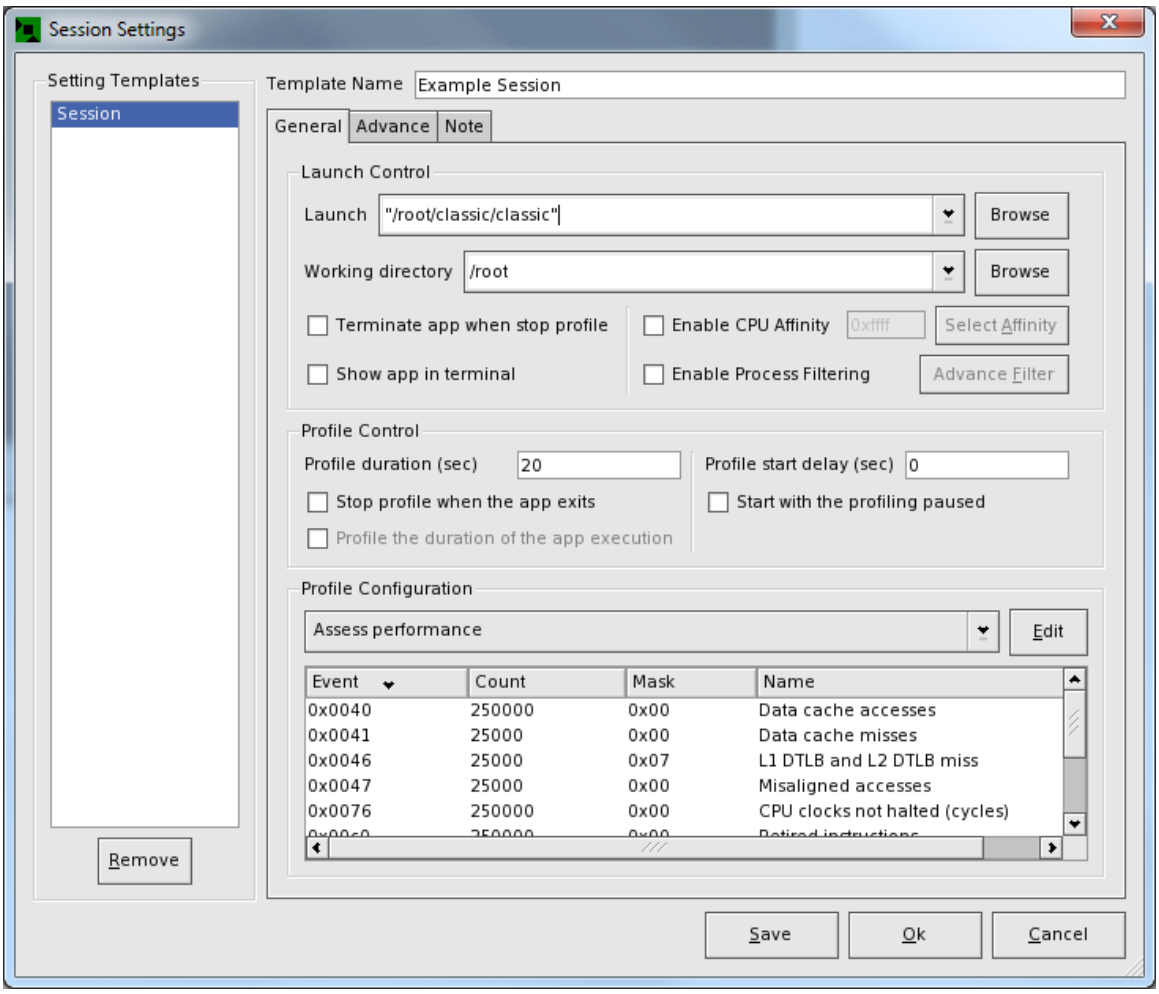
5. Click **OK** to create the new project. A dialog box appears asking for basic session settings. The session settings name the session to be created, control the kind of data to be collected, and specify the application program to be launched along with its working directory.



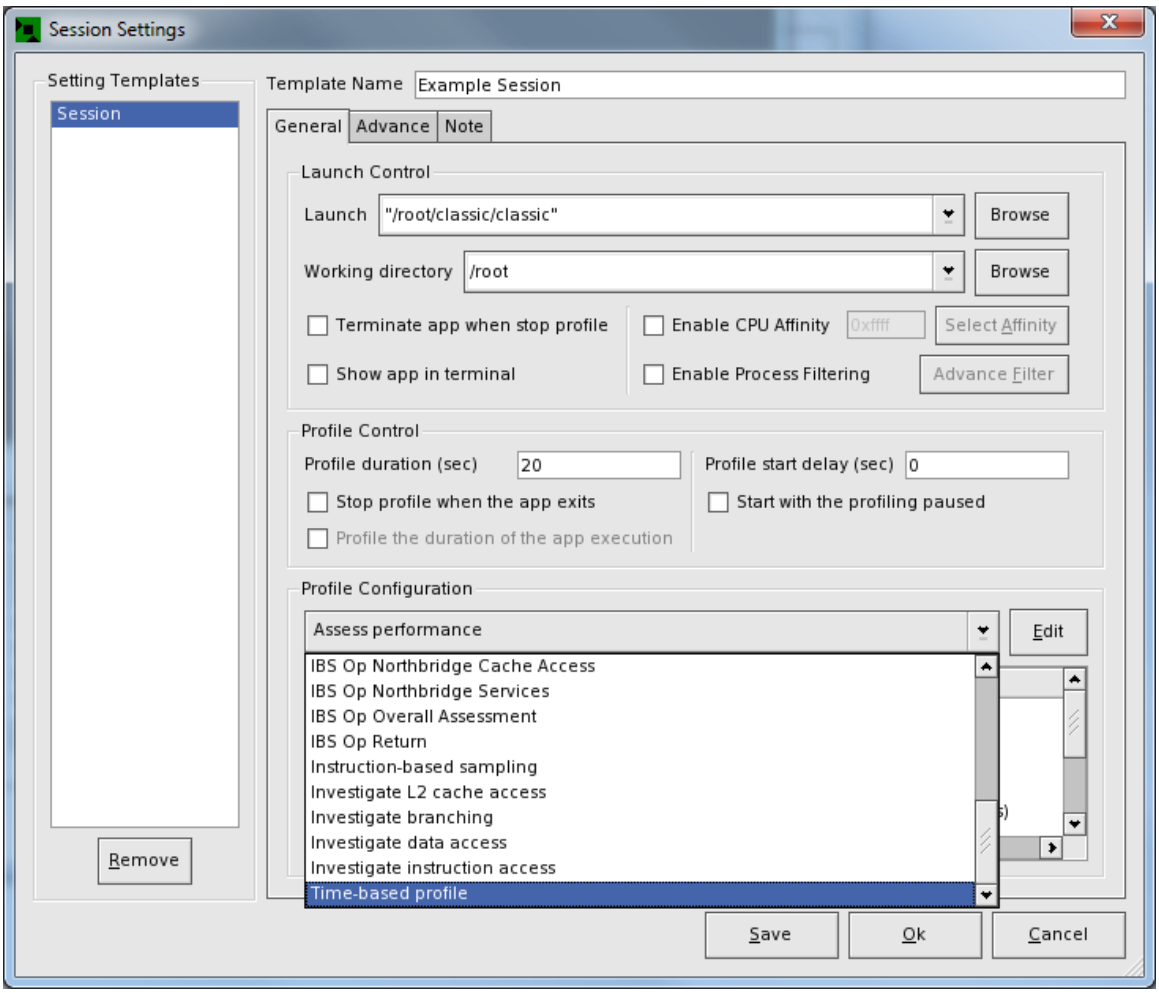
6. Enter the name of the new session into the **Template Name"** field.

7. Enter the **"Launch"** path to the application program to be launched.

8. Enter the path to the **"Working directory"**. You may browse to either the application program or working directory by clicking the appropriate **Browse** button. This tutorial uses the example program that is installed with CodeAnalyst. The profile configuration controls the method of data collection and the kind of performance data to be collected.



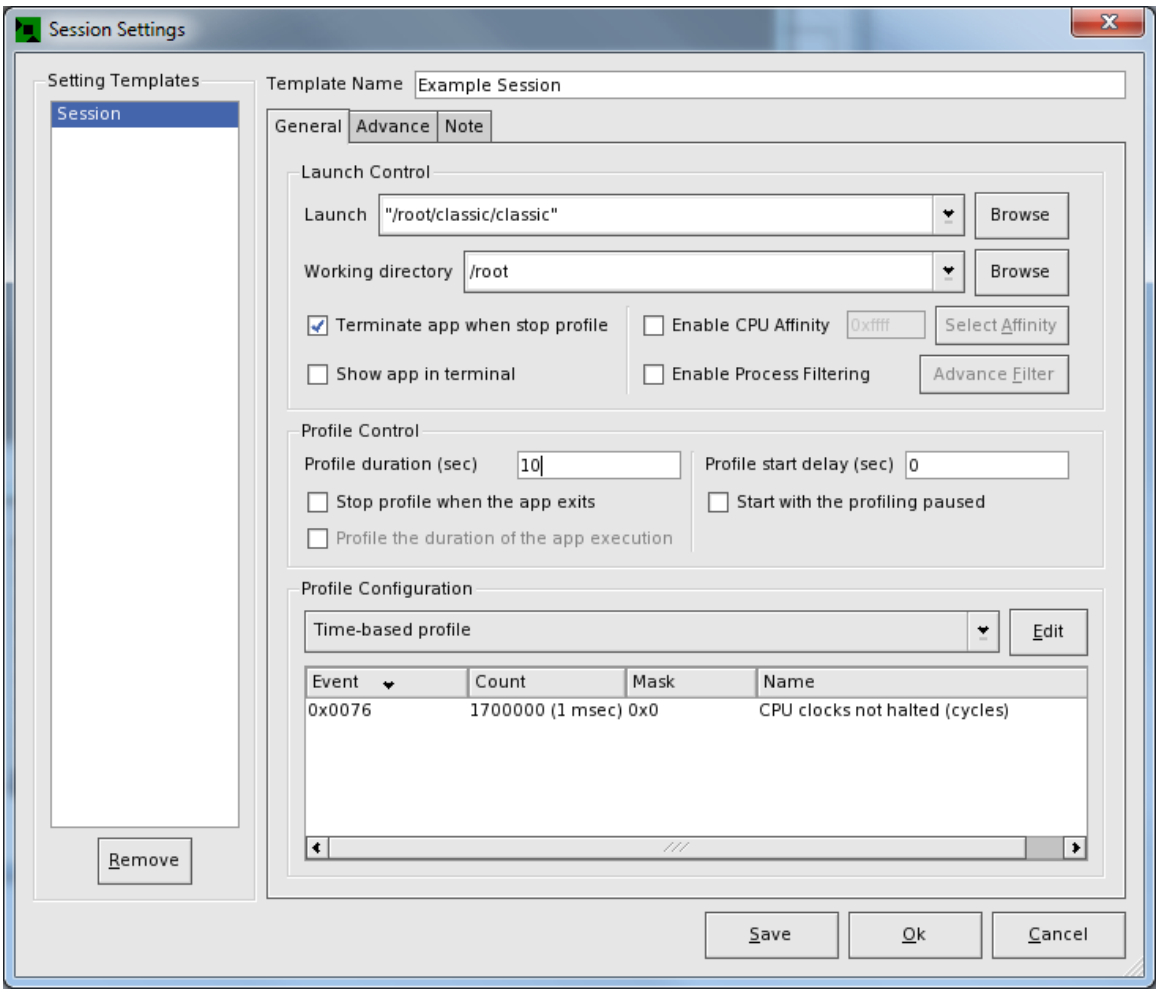
9. Select **Time-based profile** (or another profile configuration) from the drop-down list of profile configurations. A brief description of the profile configuration is displayed from the selected configuration.



10.Click the check box to the left of **"Terminate the app after the profile"** to select that option. When this option is enabled, CodeAnalyst terminates the application program after collecting data. The “Profile duration” option specifies the maximum time period for data collection.

11.Enter **"10"** into the **"Profile duration"** field to collect data for a maximum of 10 seconds.

12.Click **"OK"** to confirm the session settings. CodeAnalyst is now ready to collect data.

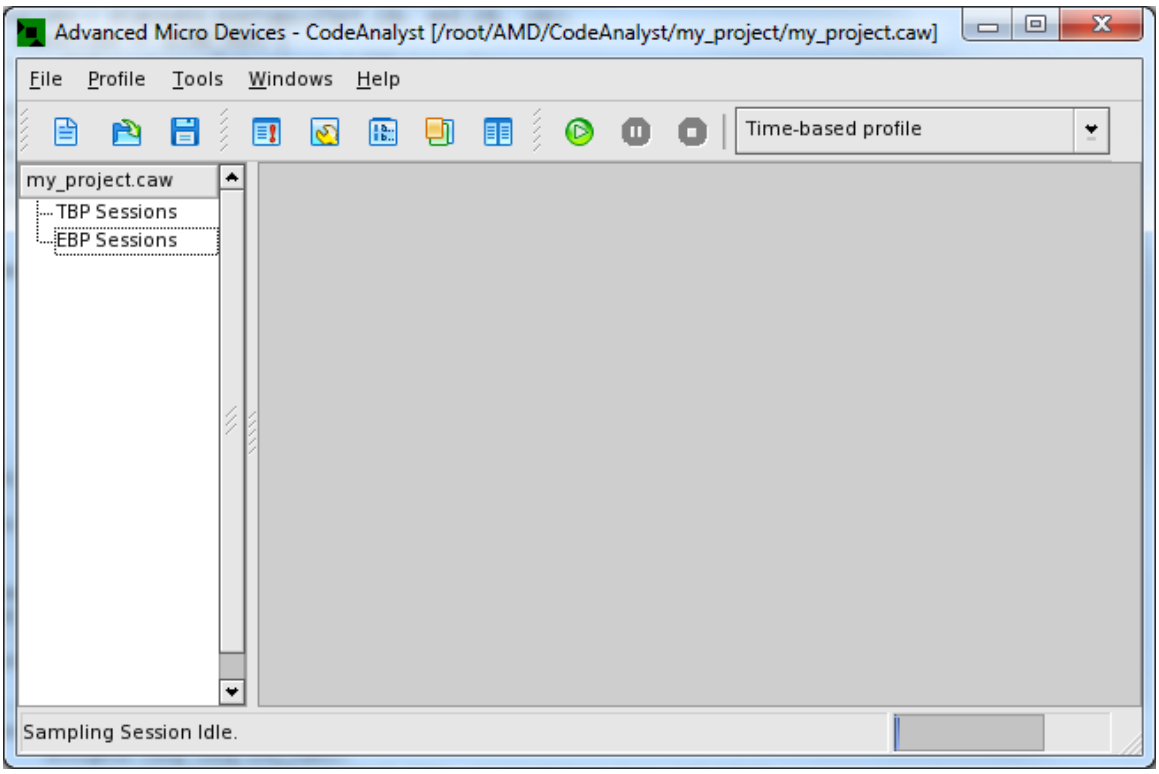


13. After the new project is created, it becomes the current project with the new project name displayed in the sessions area on the left side of the CodeAnalyst window. No sessions are listed under TBP Sessions, EBP Sessions, or IBS Sessions until data has been collected and processed.

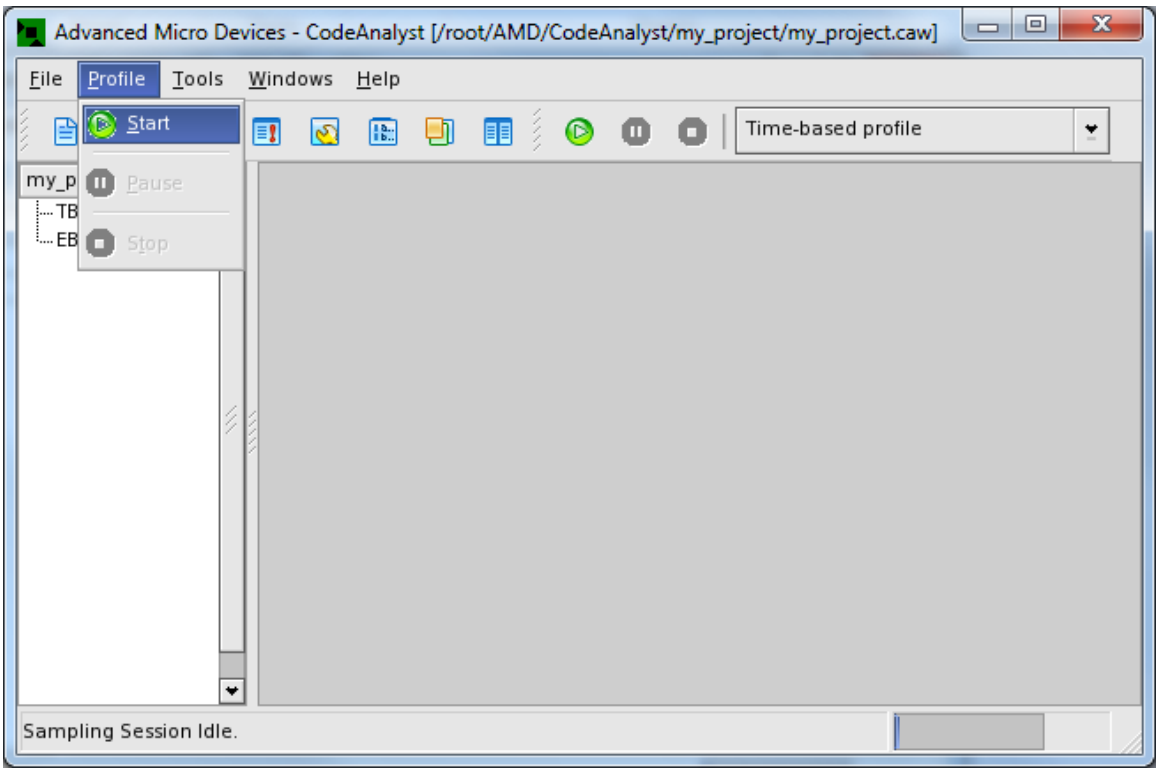
14. Click on **"Session Settings"** in the toolbar or select **"Tools > Session Settings"** to change session settings.

8.4. Tutorial - Analysis with Time-Based Sampling Profile

After creating a new project (or opening an existing project), CodeAnalyst is ready to collect and analyze data. The currently selected profile configuration is shown in the toolbar. You may choose a different profile configuration from this list without affecting the other session settings. This is a fast way to collect and analyze the program from a different perspective while keeping run control options the same. This section uses the **"Time-based profile"** configuration.



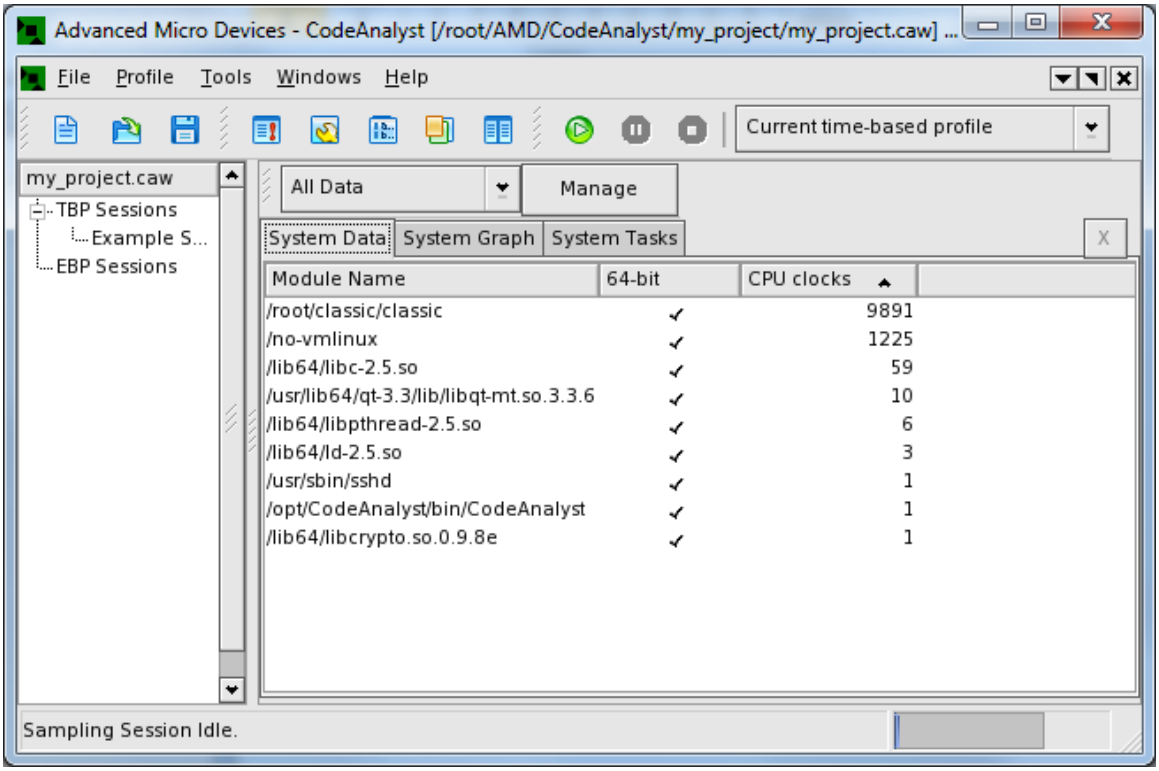
Click the **"Start"** button  in the toolbar to start data collection. You may also select **"Profile > Start"** from the Profile menu.



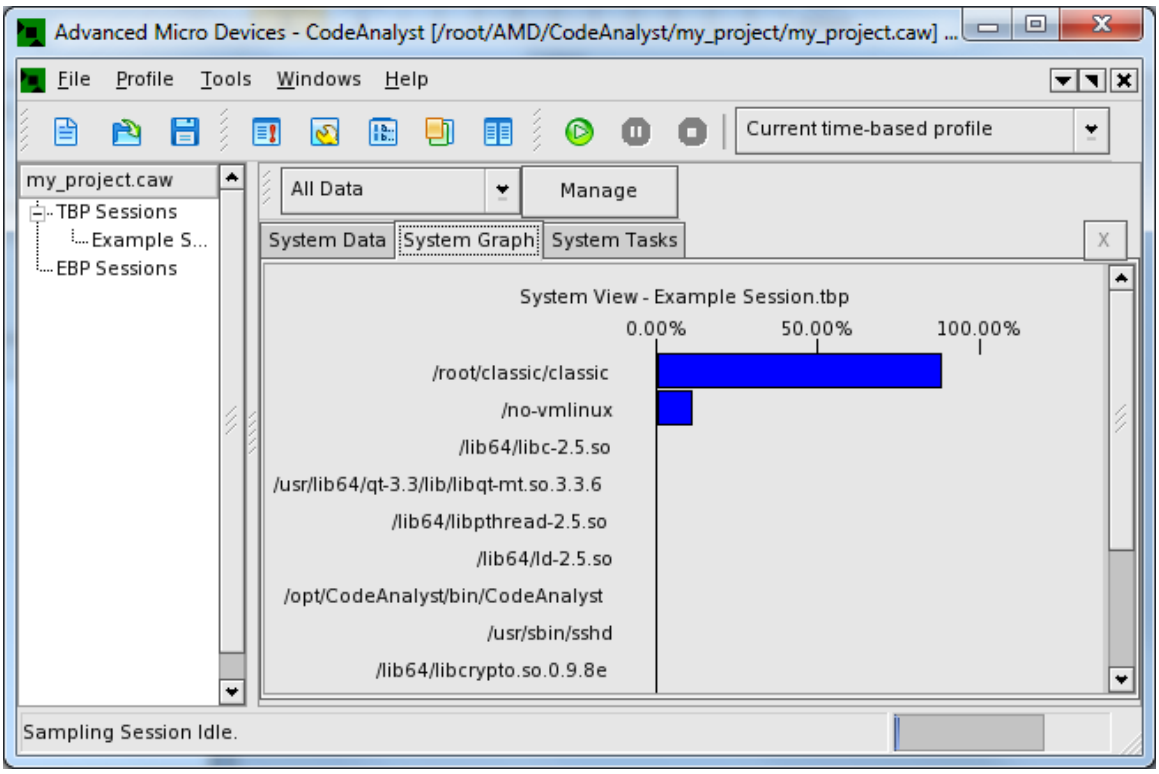
CodeAnalyst starts data collection and launches the application program that is specified in the session settings. Session status is displayed in the status bar in the lower left corner of the CodeAnalyst window. Session progress is displayed in the lower right corner.

After data collection is complete, CodeAnalyst processes the data and displays results in three tabbed panels—System Data, System Graph and Processes. A new session appears under “TBP Sessions” in the sessions area at the left-hand side of the CodeAnalyst window.

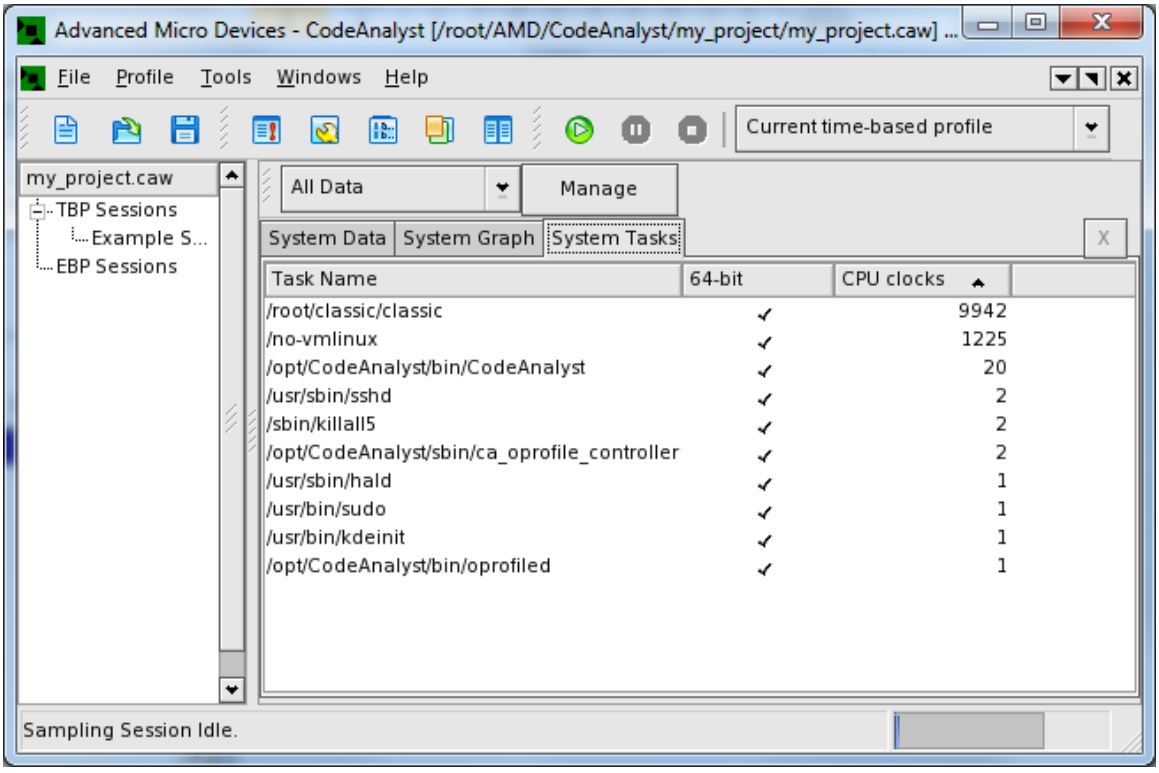
The System Data table displays a module-by-module breakdown of timer samples. This breakdown shows the distribution of execution time across modules that were active while CodeAnalyst was collecting data. CodeAnalyst is a system-wide profiling tool and it collects data on all active software components. System-wide profiling assists the analysis of multi-process applications, drivers, operating system modules, libraries, and other software components. Each sample represents 1 millisecond of execution time.



The System Graph shows the module-by-module breakdown in a bar chart. A color key is displayed below the graph. Scroll down to see the color key, if necessary. The modules that consume the most execution time are the best candidates for optimization.



The System Tasks tab displays a task-by-task breakdown of timer samples. The system-idle process is clearly identified. The example program, "**classic**", is single-threaded and fully utilizes the equivalent of only one core.



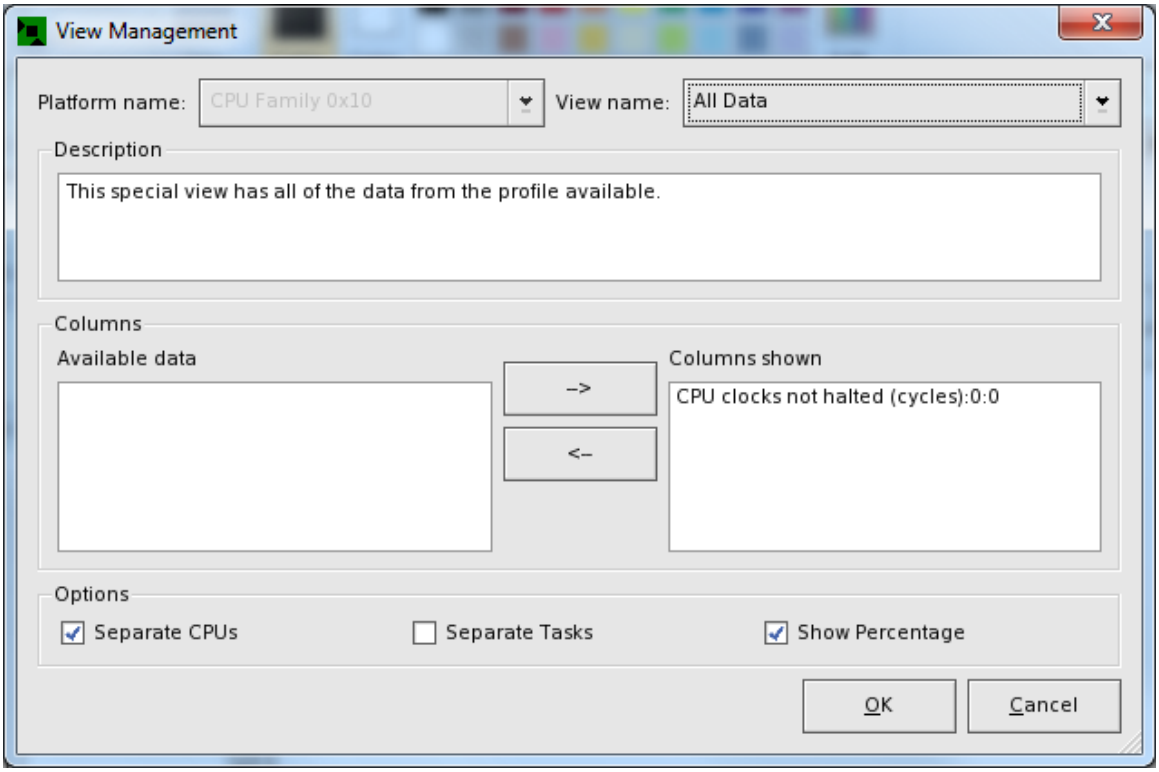
8.4.1. Changing the View of Performance Data

The CodeAnalyst GUI offers one or more views of performance data. A view shows one or more kinds of performance data or computed performance measurements that are calculated from the performance

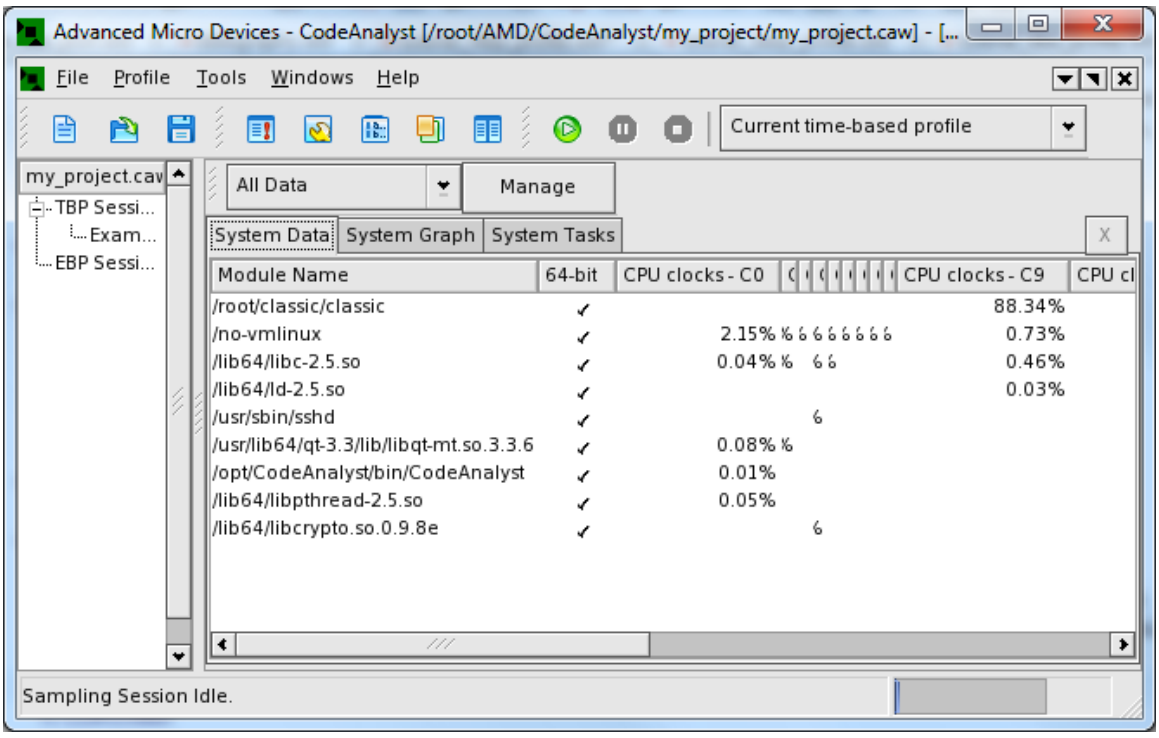
data. Select the current view from the drop-down list that appears directly above the result tabs. The “Timer-based profile” view is offered for TBP data.

1. Click the **"Manage"** button. A dialog box appears to change aspects of the currently selected view. (The tutorial section on Section 8.5, “Tutorial - Analysis with Event-Based Sampling Profile” revisits Section 7.3, “View Management”.)

   One aspect of a view is the separation and aggregation of performance data. An aggregated sample count is the sum total of the samples taken across all CPUs (processes or threads.) Aggregated data is shown by default. Data may be separated by CPU, process, or thread using the check boxes in the view management dialog box.



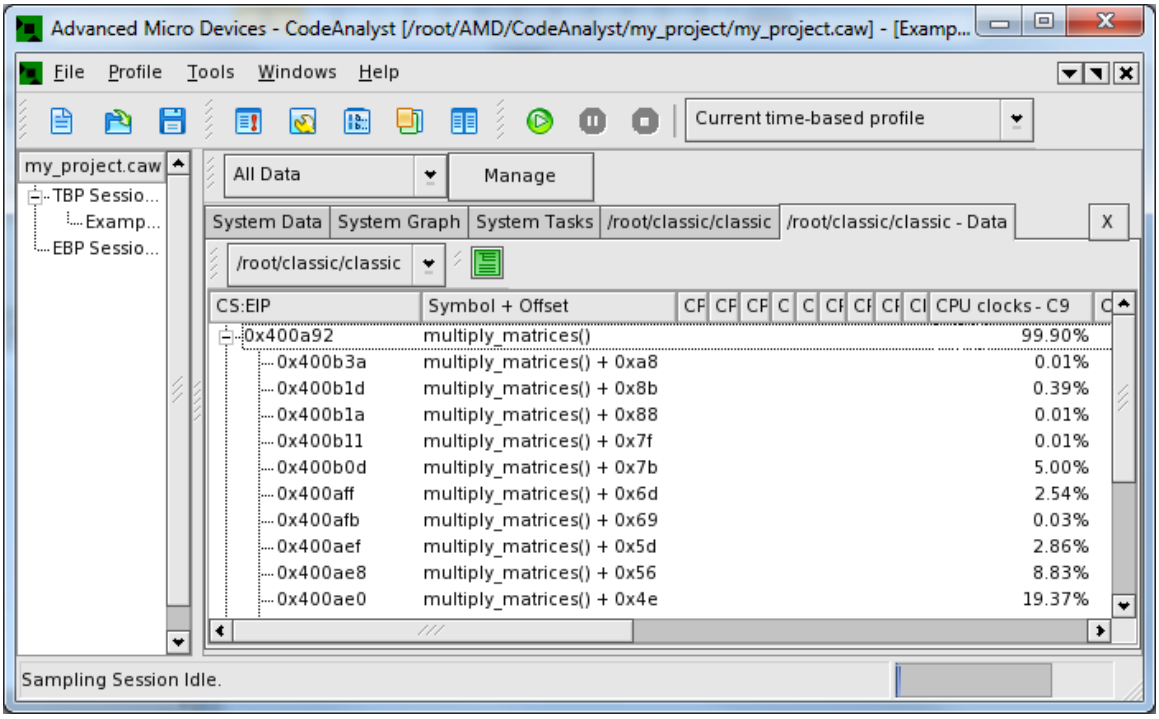
2. When the Separate CPUs option is enabled for a view, CodeAnalyst displays sample data broken out for each core. The following screen shot shows sample data for each module by individual core. The application program, **"classic"**, executed on core 9 (C9).



3. Double-click on a module in the **"System Data"** table to drill down into the data for the module.

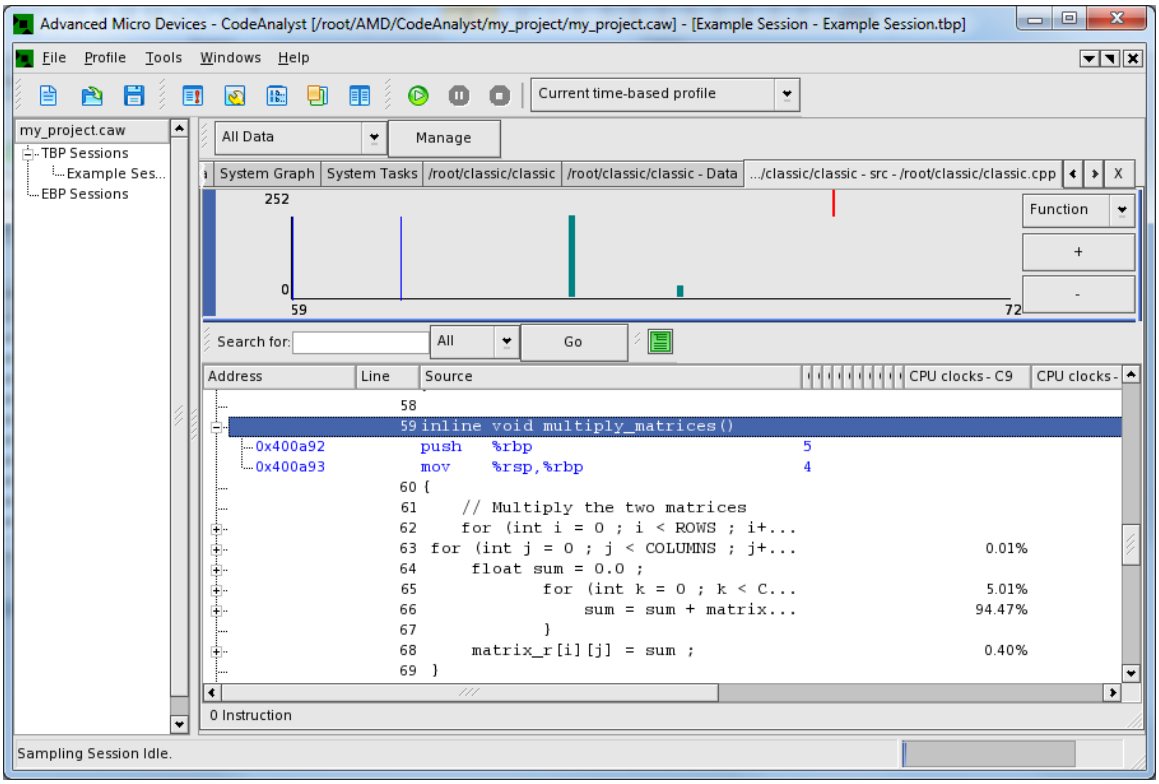
4. When viewing the **"System Graph"**, double-click on a bar in the bar chart to drill down into the corresponding module.

5. When viewing the **"System Tasks"**, double-click on a process to drill down. A new tab is created displaying a function-by-function breakdown of timer examples. The distribution of timer samples across all of the functions in the module is shown. The functions with the most timer samples take the most time to execute and are the best places to look for optimization opportunities.

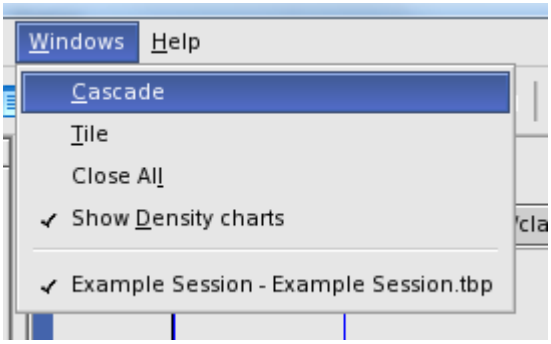


6. Double-click on a function to drill down into the data for that function. CodeAnalyst displays a new tab containing the source for the function along with the number of timer samples taken for each

line of code. A code density chart is drawn at the top of the source panel. The number of samples for each CPU is broken out since the Separate CPUs option is still enabled.

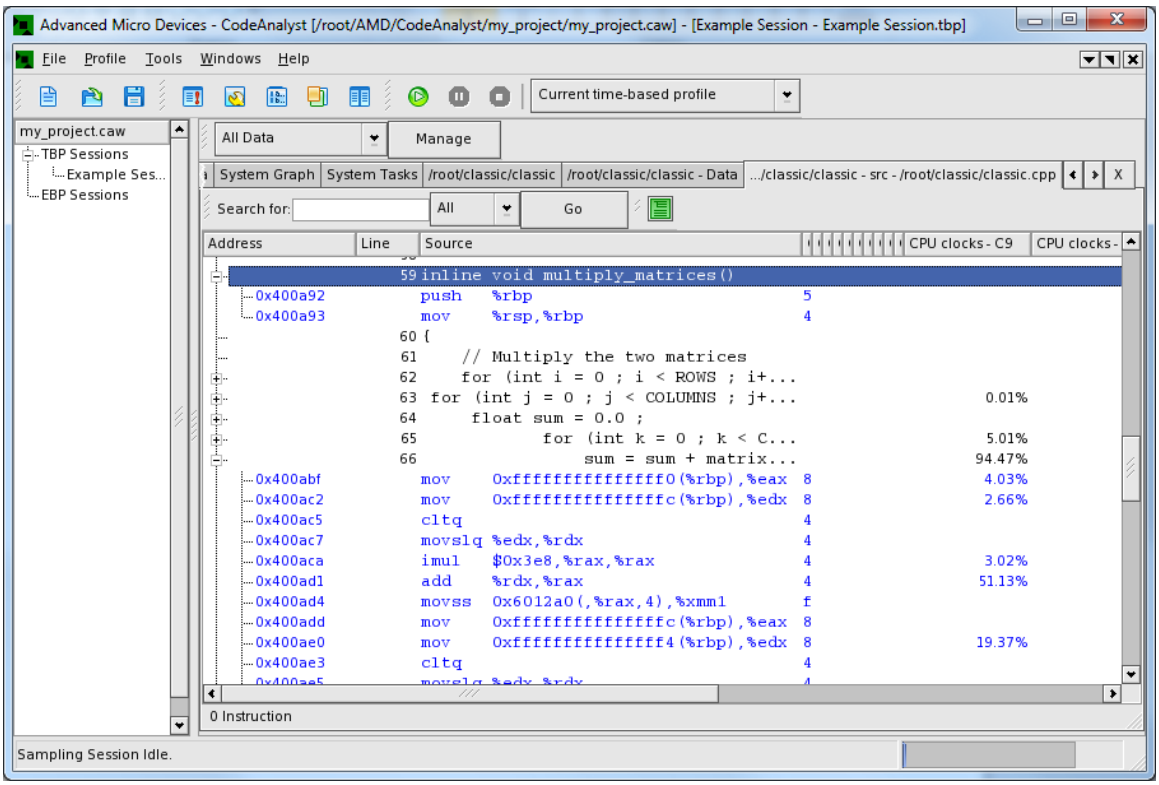


7. Some users may choose to hide the code density chart in order to devote more screen area within a source panel to code. To hide the code density chart, select **"Windows > Show Density Charts"** from the Tools menu.



8. Click on an expand (+) square to expand and display a region of source or assembly code, or click on a collapse (–) square to hide a region of source or assembly code.

   Timer samples for each source line and assembly instruction are shown. The source panel shows the program regions that are taking the most execution time. These hot-spots are the best candidates for optimization.



8.5. Tutorial - Analysis with Event-Based Sampling Profile

This section is a brief introduction to analysis with event-based profiling. A CodeAnalyst project must already be opened by following the directions under Section 8.3, “Tutorial - Creating a CodeAnalyst Project”, or by opening an existing CodeAnalyst project. It also assumes that session settings have been established and CodeAnalyst is ready to profile an application.

Event-based profiling uses the hardware performance event counters to measure the number of specific kinds of events that occur during execution. Processor clock cycles, retired instructions, data cache accesses and data cache misses are examples of events. The specific events to be measured are determined by the profile configuration that is used to set up data collection. CodeAnalyst provides five predefined profile configurations to collect performance data using event-based sampling. These profile configurations are:

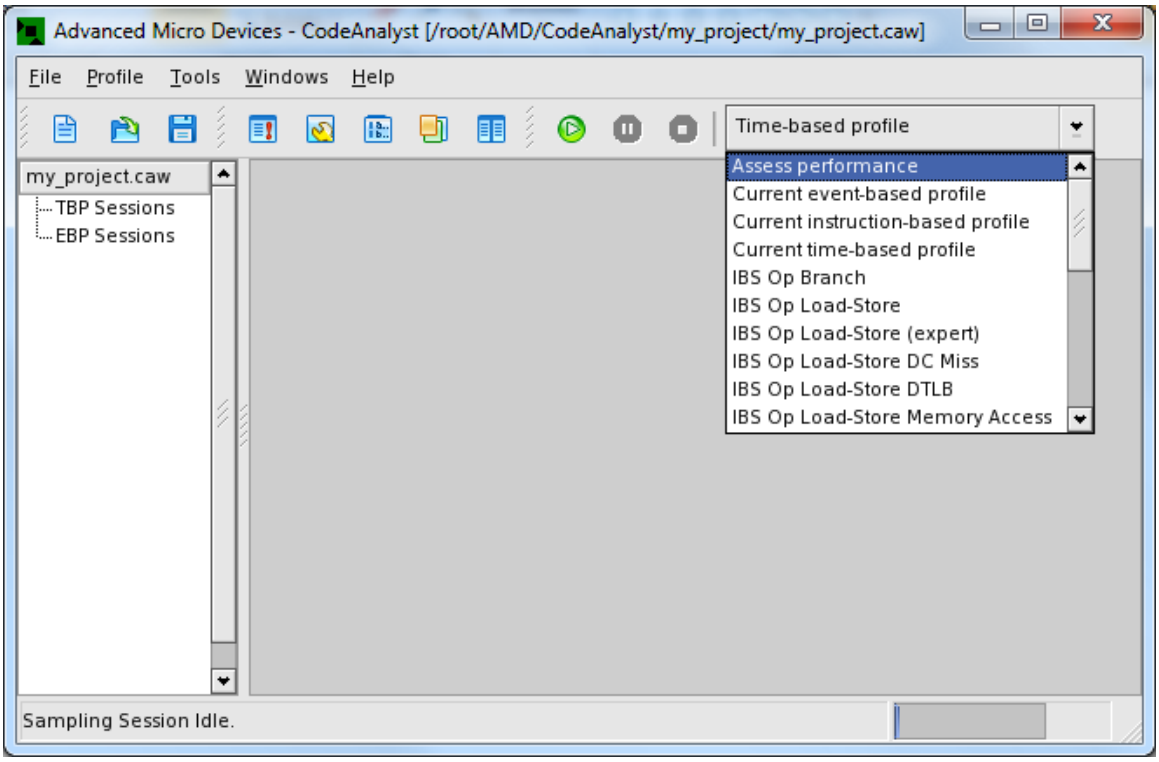
- **Assess performance**: Assess overall program performance.
- **Investigate data access**: Investigate how well software uses the data cache (DC).
- **Investigate instruction access**: Investigate how well software uses the instruction cache (IC).
- **Investigate L2 cache access**: Investigate how well software uses the unified level 2 (L2) cache.
- **Investigate branching**: Identify mispredicted branches and near returns.

These profile configurations cover the most common program performance issues of interest. Later in this section, we demonstrate how to select events and configure data collection in order to investigate issues that are not covered by the predefined profile configurations.

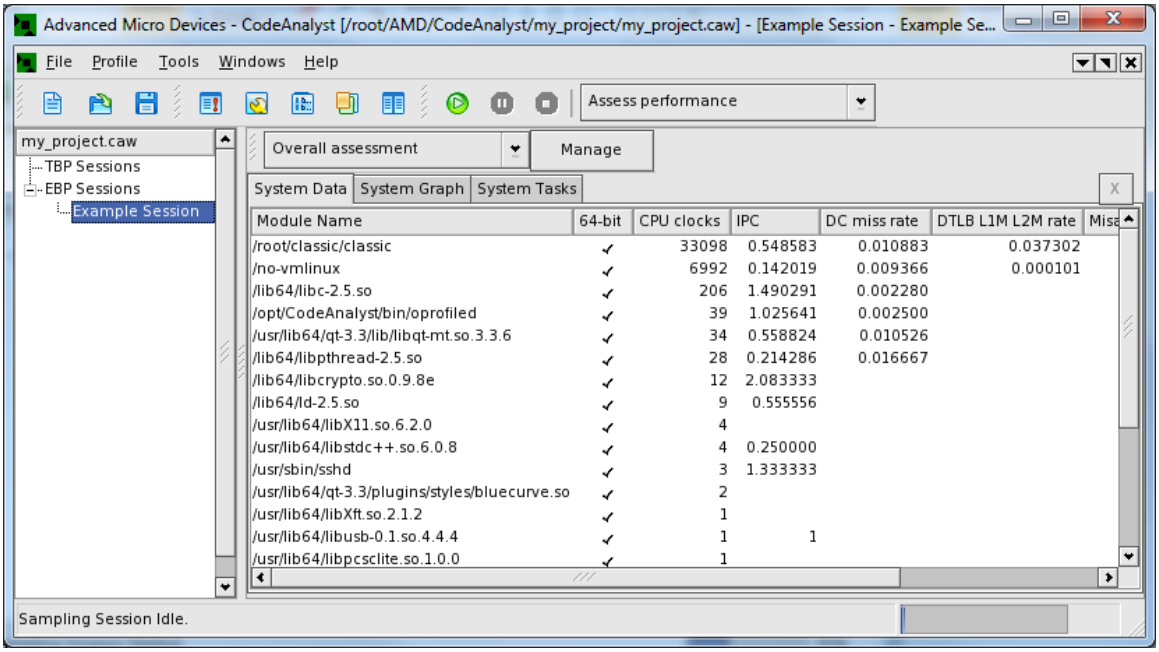
8.5.1. Assessing Performance

A drop-down list of the available profile configurations is included in the CodeAnalyst toolbar.

1. Select the **Assess performance** profile configuration. This profile configuration is a good starting point for analysis because it generates an overview of program performance. The overall assessment may indicate one or more potential issues to be investigated in more detail by using one of the other predefined configurations (or by using a custom profile configuration of your own).



2. Click the **Start** button in the toolbar or select **Profile > Start** to begin profiling. CodeAnalyst starts data collection and launches the application program that was specified in the session settings. The session status displays in the status bar in the lower left corner of the CodeAnalyst window. Session progress displays in the lower right corner. The blank window is the console window in which the application program, “classic” is running.



When data collection is complete, CodeAnalyst processes the performance data and creates a new session under “EBP Sessions” in the session management area on the left side of the CodeAnalyst

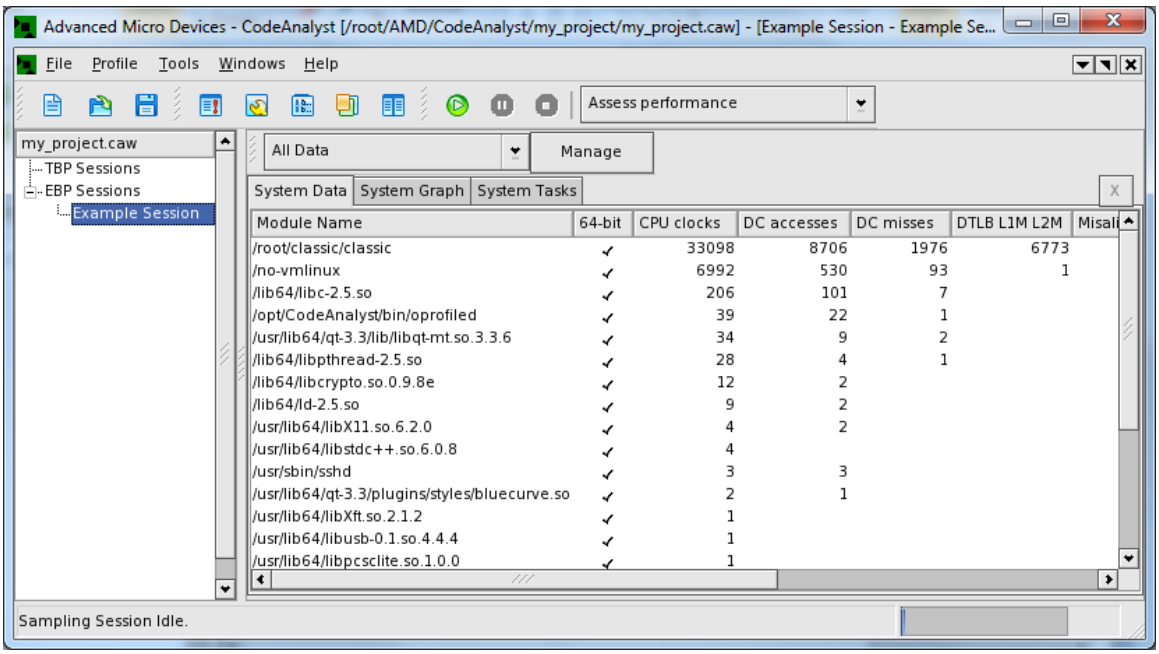
window. Results are shown in the System Data, System Graph and Processes tabs. The System Data table, System Graph and Processes table resemble and behave like their TBP counterparts. However, the type and number of event-based samples are shown instead of timer samples.

The **Overall assessment** view displays an overview of software performance. The System Data table shows the number of events and computed performance measurements for each module that was active during data collection. The Overall assessment view shows:

- CPU clocks: CPU clocks not halted event
- Instructions per clock cycle (IPC): Retired instructions / CPU clocks
- DC (data cache) miss rate: DC misses / retired instructions
- DTLB (data translation lookaside buffer) miss rate: DTLB miss / retired instructions
- Misaligned access rate: Misaligned accesses / retired instructions
- Mispredict rate: Retired mispredicted branch instructions / retired instructions

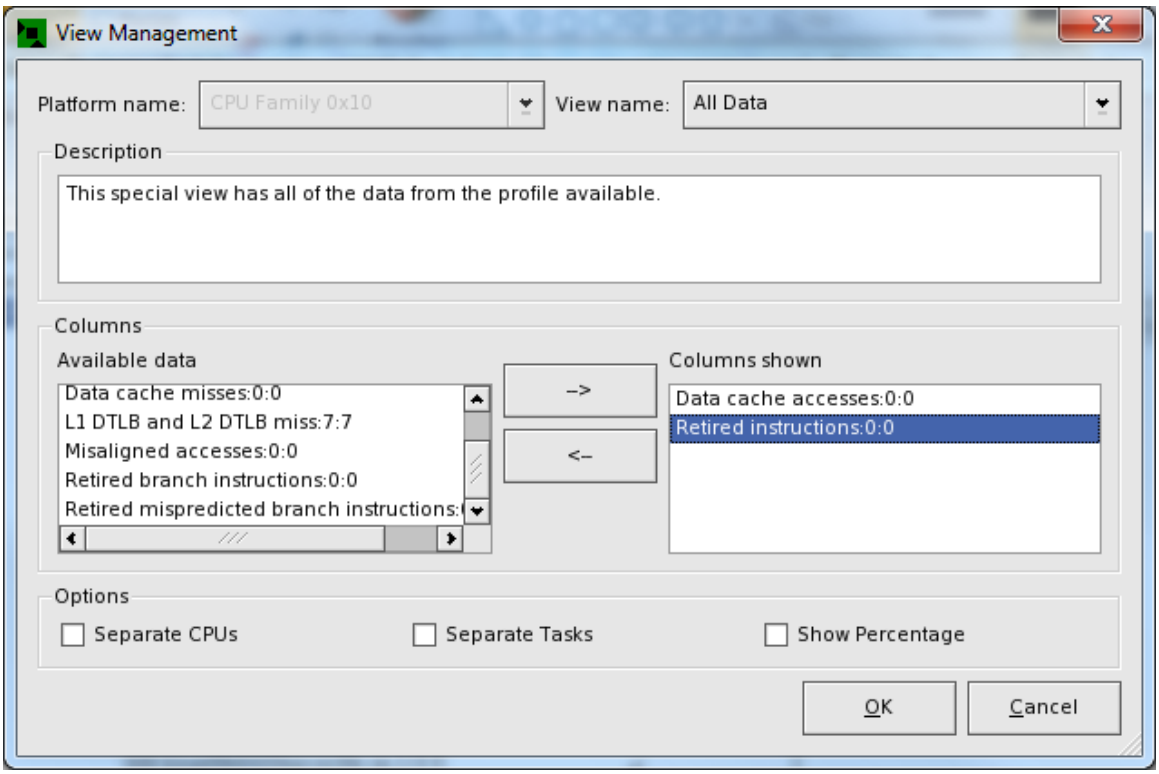
In general, when the term **rate** appears in a computed performance measurement, the rate is expressed as “events per retired instruction.” A rate indicates how frequently an event is occurring. A high rate, such as a high DC miss rate, may indicate the presence of a performance problem and an opportunity for optimization.

The specific combination of events and computed performance measurements that are shown in a table or graph are a **view**. CodeAnalyst may offer more than one view depending upon the kinds of data (e.g., events) that were collected. The drop-down list (immediately above the System Data tab) contains the available views. The All Data view is always available. To switch to All Data view, select the **"All Data"** view from the drop-down list.



8.5.2. Changing Contents of a View

CodeAnalyst provides a way to change the contents of a view.



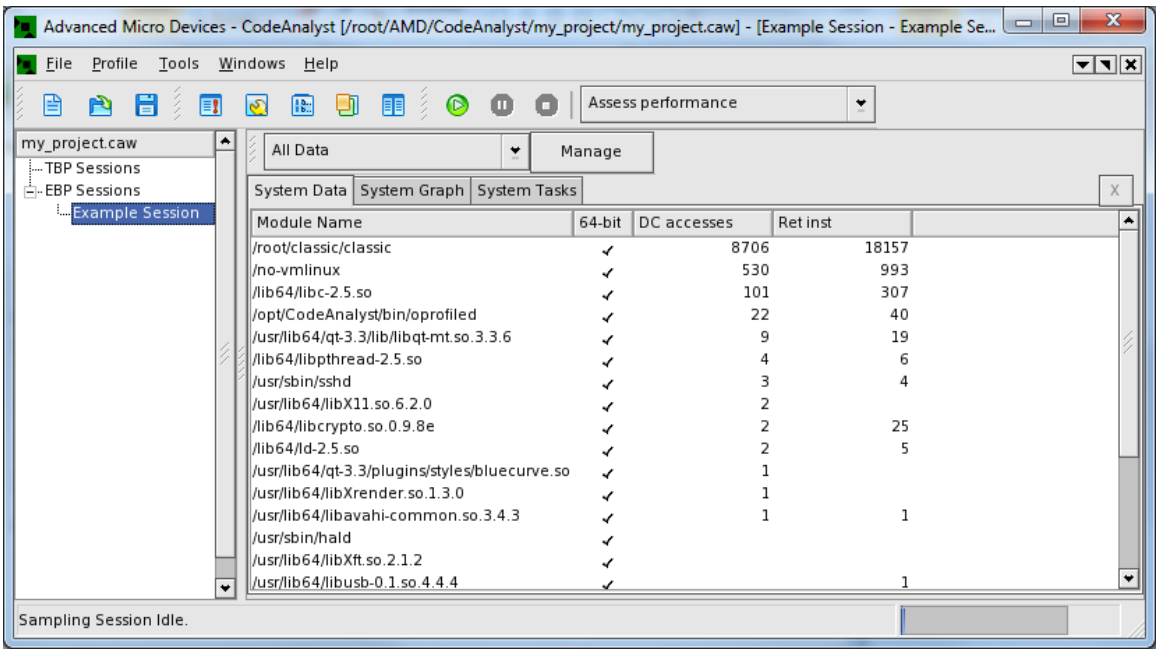
1. Click the **Manage** button to change the contents of the currently selected view. A dialog box appears showing the name of the view, a description of the view, the available data that can be shown and the columns (data) that are shown.

   - To add data for an event to the current view, select an event in the **Available data** list and click the **right arrow** button.

   - To remove data for an event from the current view, select an event in the **Columns shown** list and click the **left arrow** button.

2. Remove all events except **Retired instructions** and **Data cache assesses** from the Columns shown list.

3. Click the **OK** button to confirm and accept the changes. After making these changes, CodeAnalyst updates the System Data table and eliminates the columns for the event data that were removed from the view.

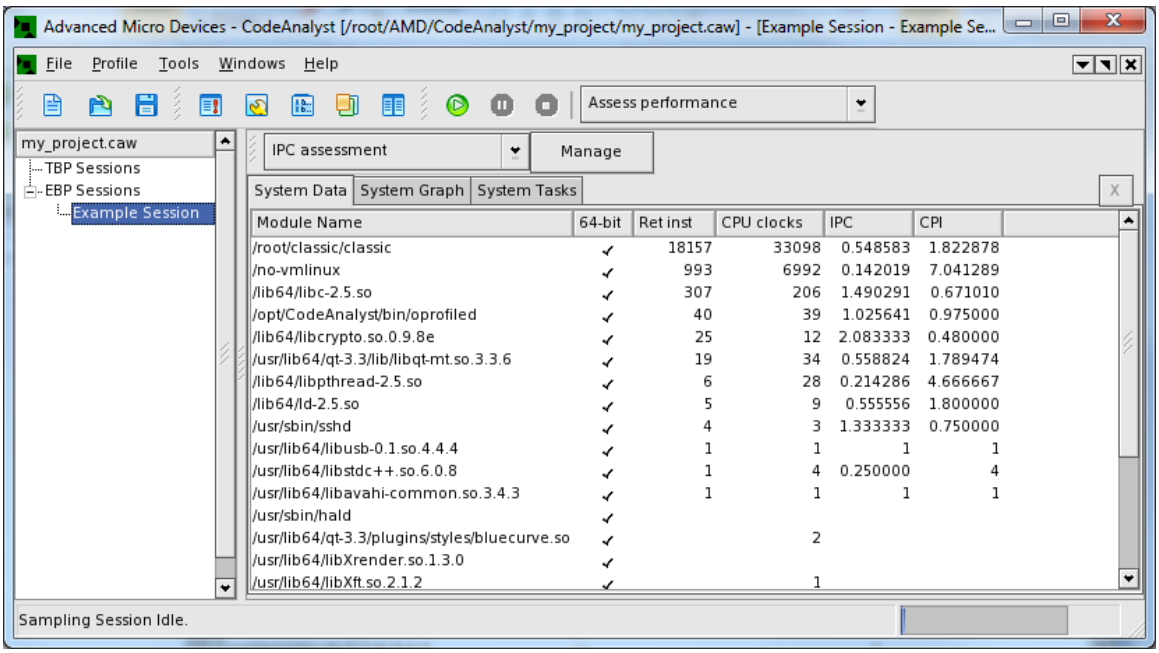


4. Select the **IPC assessment** view from the drop-down list of views.

   CodeAnalyst updates the System Data table which now shows the IPC assessment view. This view consists of:

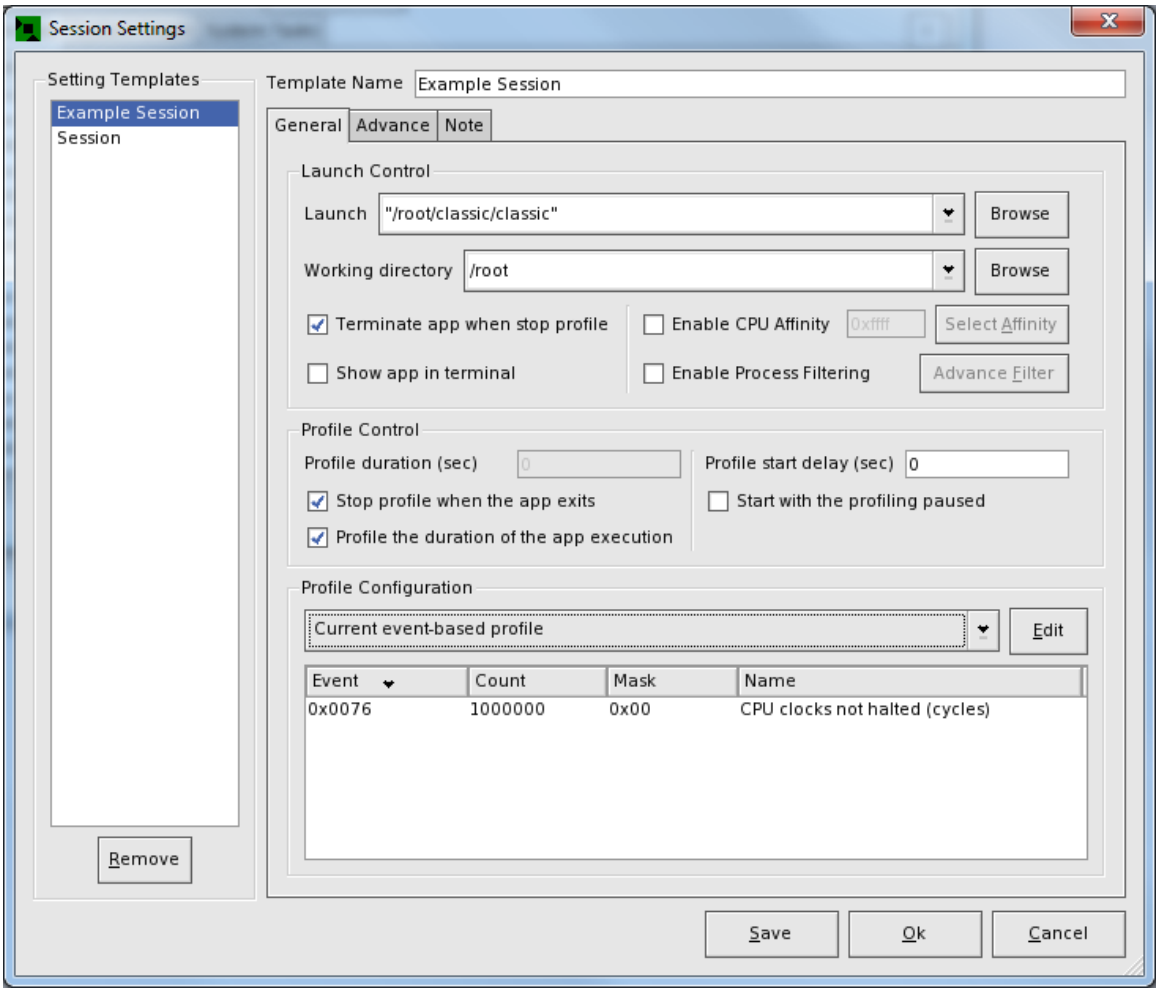
   - The number of retired instruction samples
   - The number of CPU clock samples
   - The ratio of instructions per clock cycle (IPC)
   - The ratio of clock cycles per instruction (CPI)

The ratio of instructions per clock cycle is a basic measure of execution efficiency and is a good indicator of instruction level parallelism (ILP).



8.5.3. Choosing Events for Data Collection

The predefined profile configurations cover the most common kinds of performance analysis. AMD processors, however, are able to monitor a wide range of performance events.



1. To configure data collection using events of your own choice, click on the **"Session Settings"** button in the toolbar. A dialog box appears asking for session settings.

2. Choose the **Current event-based** profile configuration in the list of profile configurations. You may freely edit and change this profile configuration and may use this profile configuration as a scratchpad for customized EBP configurations.

3. Click the **Edit** button. A dialog box appears which allows you to edit the “Current EBS/IBS Profile Configuration" dialog. The “Current event-based profile” configuration in this example already contains the “CPU clocks not halted” event.

4. In "Perf Counter Tab", scroll through the list of individual events to find the **"Retired instructions"** event. Select **"Retired instructions"** and click **"Add Event"**. The “0x00c0 Retired instructions” event is added to the list of events in the configuration. Set the **"Count"** field to **"1,000,000"**, and click **"Apply Setting"**.

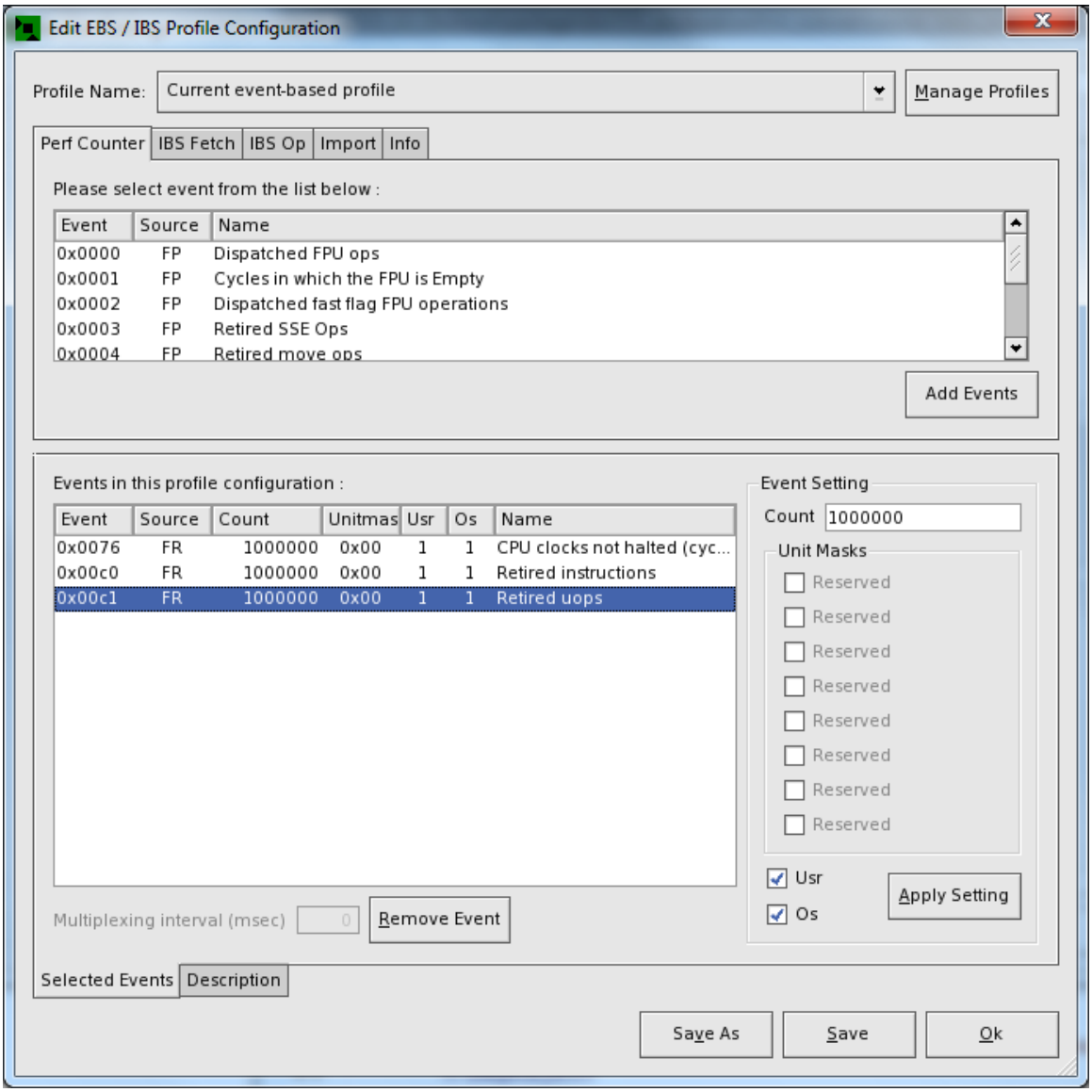
5. Find and select the **"0x00c1 Retired uops"** event and repeat the previous step.

If you make a mistake and need to remove an event from the configuration, select the event and then click the **"Remove Event"** button.

The Event Count field specifies the sampling period for the event. The Event Count determines how often a sample is taken for the event. If the Event Count is N, then a sample will be taken after the occurrence of N events of that type. Use smaller Event Count values to sample an event more often. However, more frequent sampling increases measurement overhead.

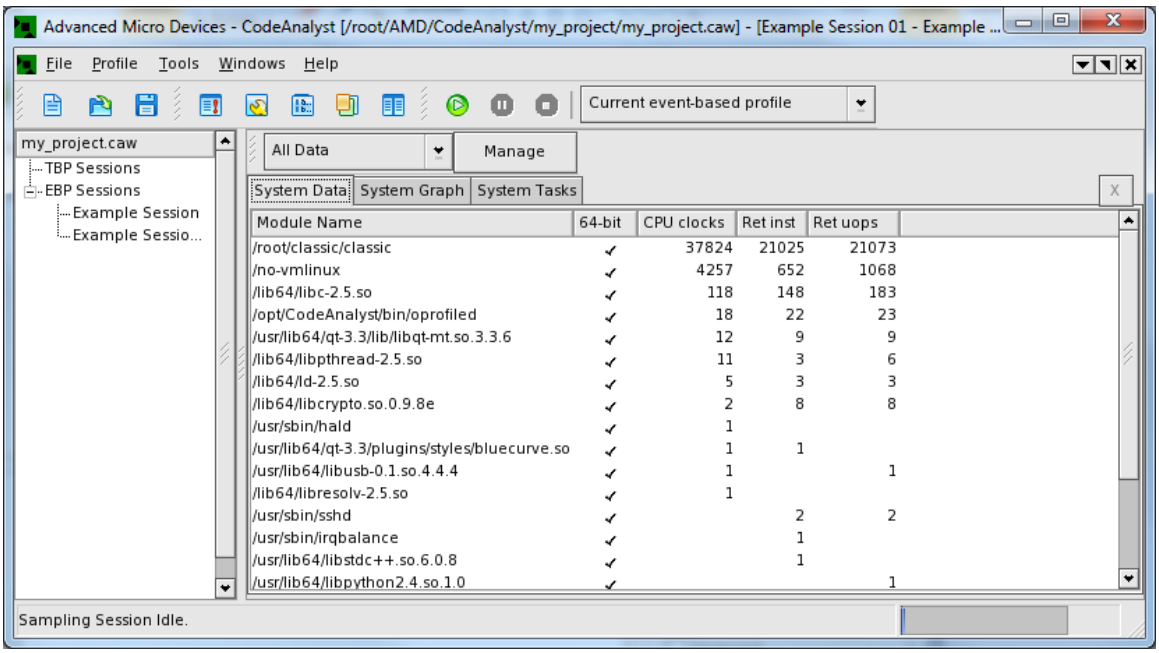
Caution: Choose the Event Count value conservatively. Start with a large value first and then decrease the value until the desired measurement accuracy is achieved. Very small values may cause the system to hang under certain workload conditions.

6. Click **OK** to confirm the changes and to dismiss the dialog box.



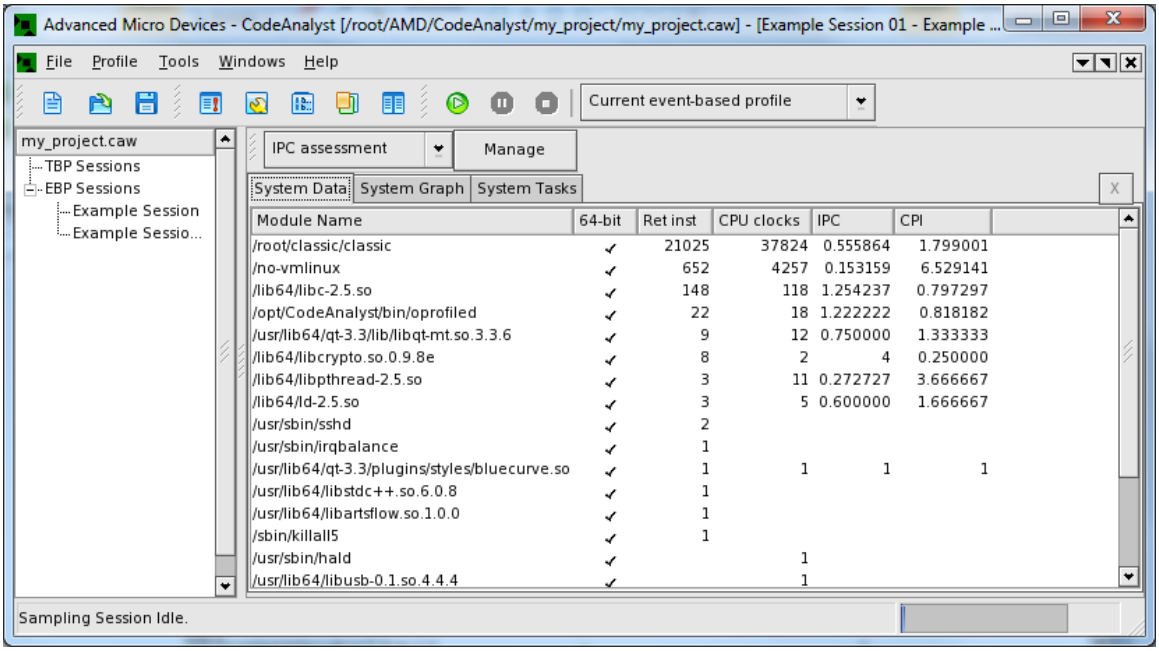
7. Click the **Start** button in the toolbar to begin data collection, or select **Profile > Start** from the Profile menu. Results are displayed in the System Data, System Graph and Processes tabs when data collection is finished. A new session, “ExampleSession1,” is added to the list of EBP sessions in the session management area. Notice that CodeAnalyst auto-generates new session names when necessary.

8. Click on the **System Data** tab and select the **All Data** view from the list of available views. Three columns display containing the number of samples taken for the CPU clocks (not halted), retired instruction, and retired micro-op (uops) events.



Examine the list of available events. The predefined **IPC assessment** view is offered because data is available for both the retired instruction and CPU clocks not halted events. The decision to offer a view is **data-driven**. If the right type of event data is available to display a view, CodeAnalyst offers the view.

9. Select the **"IPC assessment"** view to display a module-by-module breakdown of IPC and CPI measurements in the System Data table.

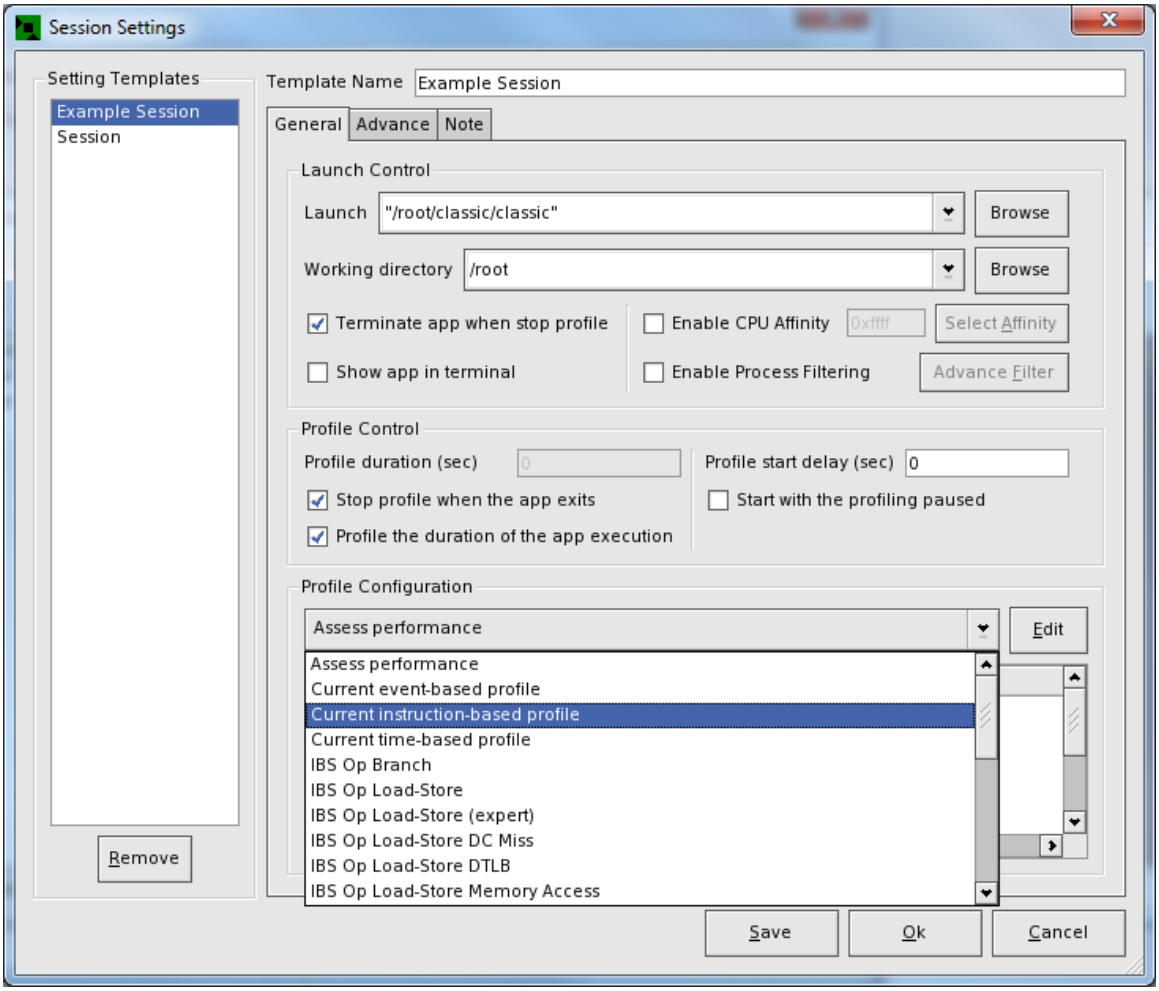


8.6. Tutorial - Analysis with Instruction-Based Sampling Profile

This section is a brief introduction to using Instruction-Based Sampling (IBS). A CodeAnalyst project must already be opened by following the directions under Section 8.3, “Tutorial - Creating a Code-

Analyst Project", or by opening an existing CodeAnalyst project. It also assumes that session settings have been established and CodeAnalyst is ready to profile an application.

8.6.1. Collecting IBS Data

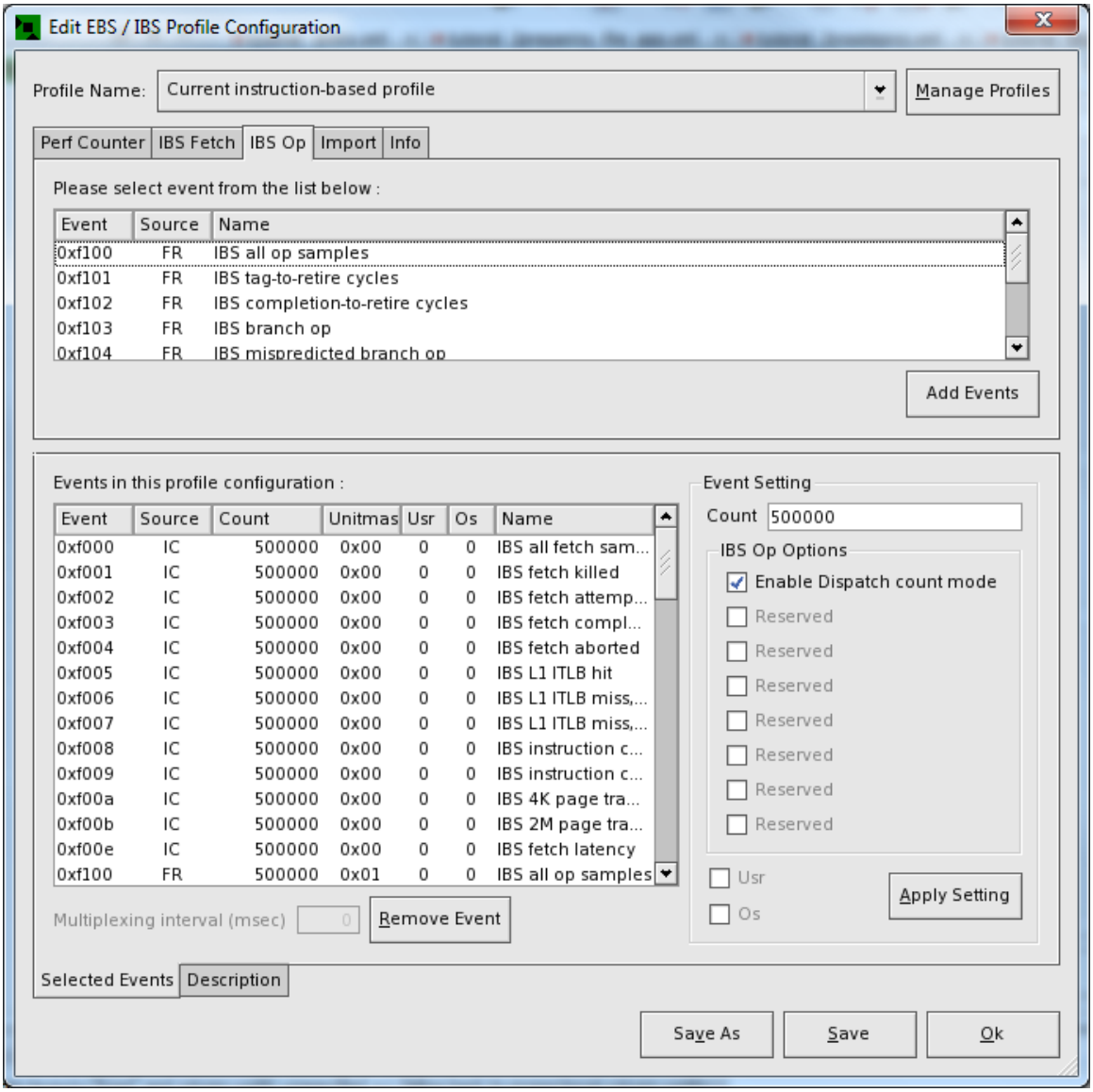


1. Open "Session Setting" dialog from the toolbar or **"Tools > Sesssion Settings"**. In the dialog, select the **"Current Instruction-based sampling"** profile configuration and click **"Edit"**

2. In the "Edit EBS/IBS Profile Configuration" dialog, there might be some events in the field "Events in this profile configuration". Remove all existing events by selecting each event and click "Remove Event" button.

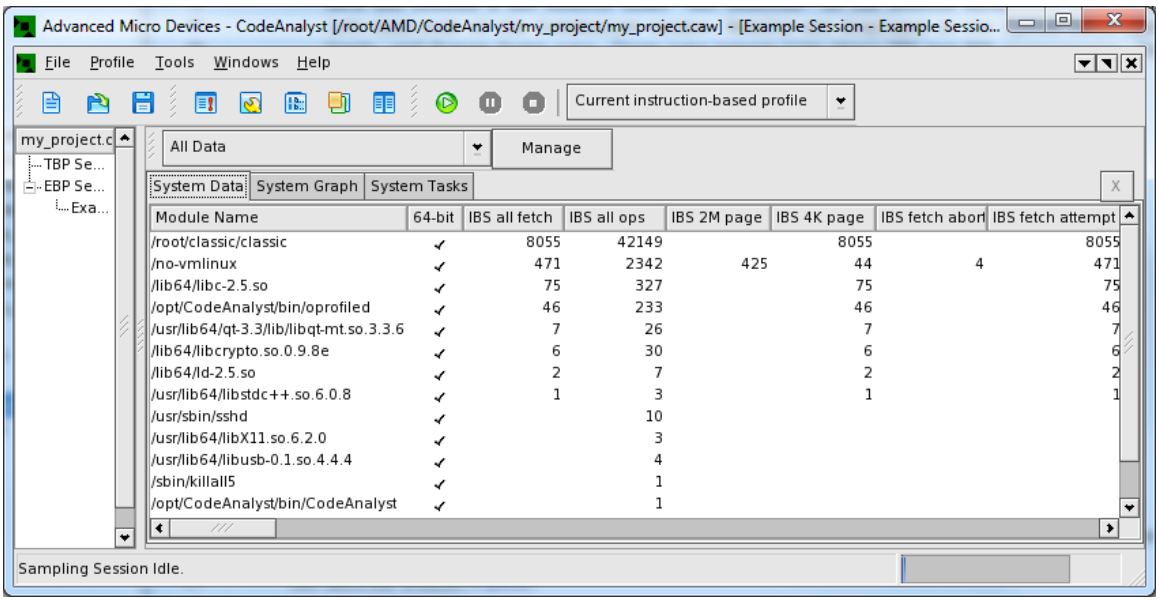
3. In the **"IBS Fetch"** tab, select all events by pressing **"Ctrl+a"** on the keyboard, then click the "Add Events" button.

4. In the **"IBS Op"** tab, select all events by pressing **"Ctrl+a"** on the keyboard, then click the "Add Events" button.



5. Click the **"Start"** button in the toolbar or select **Profile > Start** to begin profiling. CodeAnalyst starts data collection and launches the application program previously specified in the session settings. The session status displays in the status bar in the lower left corner of the CodeAnalyst window. Session progress displays in the lower right corner. The blank window is the console window in which the application program “classic” is running.

   When data collection completes, CodeAnalyst processes the IBS performance data and creates a new session under "EBP Sessions" in the session management area at the left-hand side of the CodeAnalyst window. Results are displayed in the System Data, System Graph, and System Tasks tabs. These tabs behave like their TBP and EBP counterparts. The tables and graph display the number of IBS-derived events that were sampled by the performance monitoring hardware.



8.6.2. Reviewing IBS Results

CodeAnalyst reports IBS performance data as **"IBS-derived events"**. See Instruction-Based Sampling-Derived Events for descriptions of the IBS-derived events.

Although IBS derived events look similar to performance monitoring counter (PMC) events, the sampling method is quite different. PMC event samples measure the actual number of particular hardware events that occurred during the measurement period. IBS derived events report the number of IBS fetch or op samples for which a particular hardware event was observed. Consider the three IBS derived events:

- IBS all op samples
- IBS branch op
- IBS mispredicted branch op

The **"IBS all op samples"** derived event is a count of all IBS op samples that were taken. The **"IBS branch op"** derived event is the number of IBS op samples where the monitored macro-op was a branch. These samples are a subset of all the IBS op samples. The **"IBS mispredicted branch op"** derived event is the number of IBS op sample branches that mispredicted. These samples are a subset of the IBS branch op samples. Unlike PMC events that count the actual number of branches (event select 0x0C2), it would be incorrect to say that the **"IBS branch op"** derived event reports the number of branches. The sampling basis is different.

The "All Data" view shows the number of occurrences of all IBS derived events. Instruction-Based Sampling collects a wide range of performance data in a single run. For instance, when all dervied events from both IBS fetch and op data are collected, the "All Data" view displays information for over 60 IBS derived events. However, this is not recommended since it could potentially introduce overhead when processing the profile data. CodeAnalyst provides several predefined views that display IBS derived events in logically-related groups. The available views are:

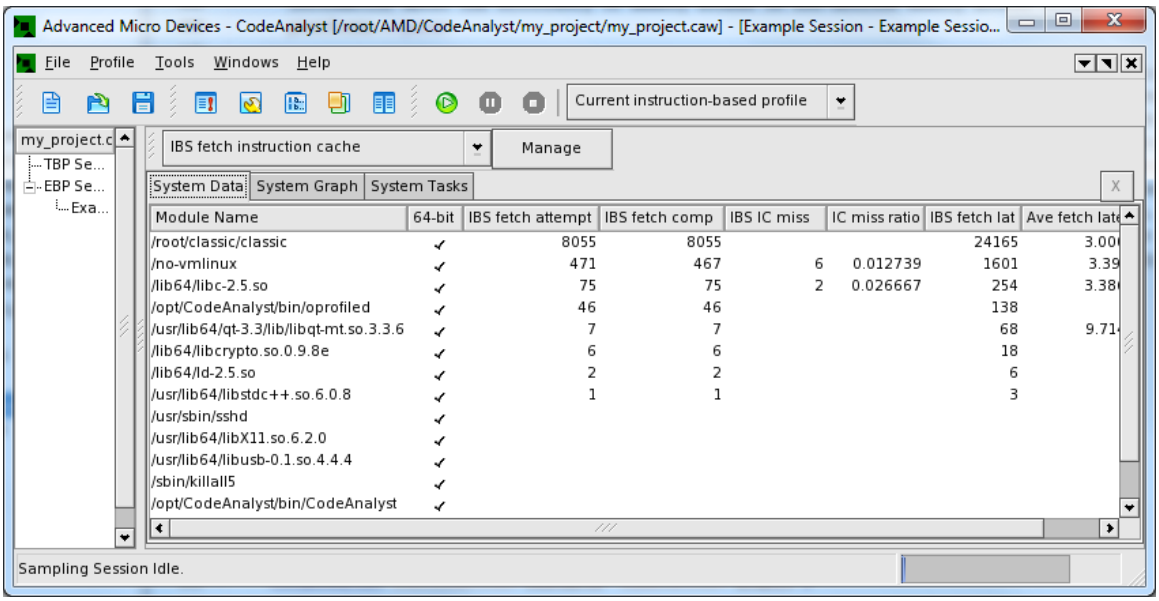
- **IBS All ops:** Breakdown of all IBS op samples by major type
- **IBS BR branch:** Detailed information about branch operations
- **IBS BR return:** Detailed information about subroutine return operations
- **IBS MEM all load/store:** Breakdown of memory access operations by major type

- **IBS MEM data TLB:** Detailed information about data translation lookaside buffer (DTLB) behavior
- **IBS MEM data cache:** Detailed information about data cache (DC) behavior
- **IBS MEM forwarding and bank conflicts:** Detailed information about store-to-load data forwarding and bank conflicts
- **IBS MEM locked ops and access by type:** Detailed information about locked operations and UC/WC memory access
- **IBS MEM translations by page size:** Breakdown of DTLB address translations by page size
- **IBS NB cache state:** Detailed information about the L3 cache state when the L3 cache services a Northbridge request
- **IBS NB local/remote access:** Breakdown of local and remote memory accesses via the Northbridge
- **IBS NB request breakdown:** Breakdown of requests made through the Northbridge
- **IBS fetch instruction TLB:** Detailed information about instruction translation lookaside buffer (ITLB) behavior
- **IBS fetch instruction cache:** Detailed information about instruction cache (IC) behavior
- **IBS fetch overall:** Breakdown of attempted, killed, completed, and aborted fetch operations
- **IBS fetch page translations:** Breakdown of ITLB address translations by page size

Most software developers will be interested in the overall breakdown of IBS ops, branch operations, load/store operations, data cache behavior, and data translation lookaside buffer behavior. The breakdown of local/remote accesses through the Northbridge can provide information about the efficiency of memory access on non-uniform memory access (NUMA) platforms.

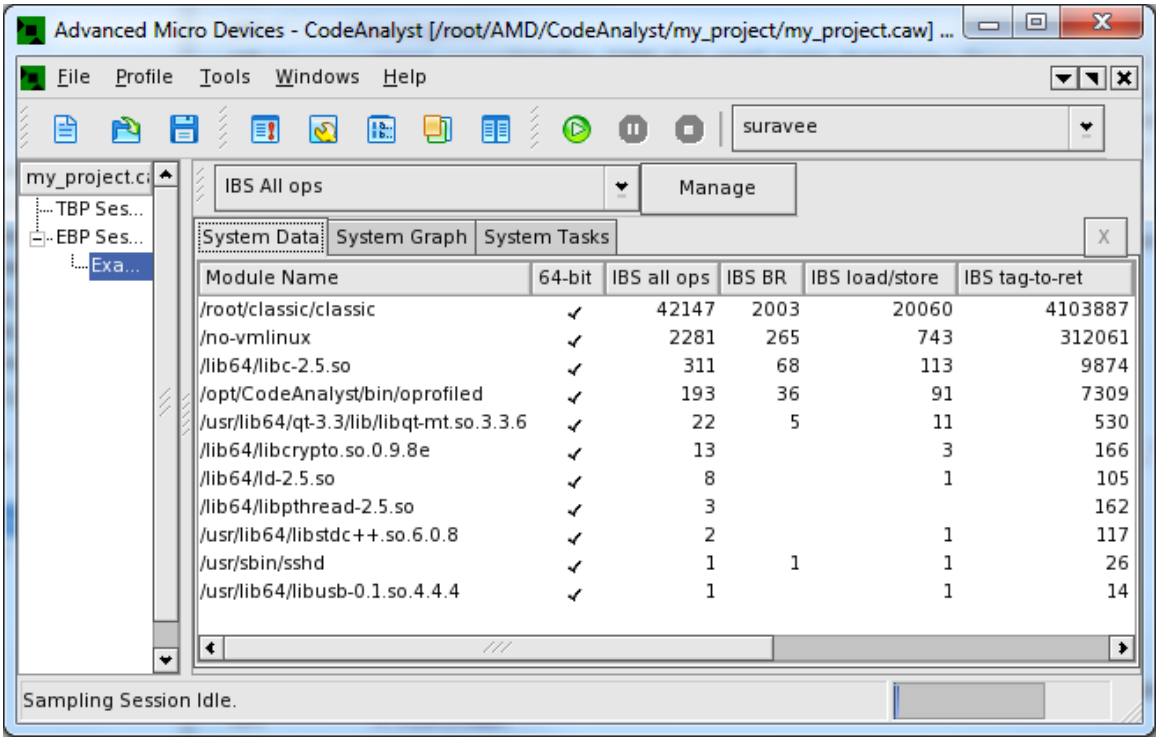
1. Select **"IBS fetch instruction cache"** from the drop-down list of views.

   IBS information about instruction cache behavior is displayed. IC-related IBS derived events are shown, including the number of IBS fetch samples for attempted and completed fetch operations, the number of fetch samples which indicated an IC miss, and the total IBS fetch latency. An **attempted** fetch is a request to obtain instruction data from cache or system memory. A fetch attempt may be speculative. A **completed** fetch actually delivers instruction data to the decoder. The delivered data may go unused if the branch operation redirects the pipeline at a later time. Finally, the view also includes two computed performance measurements—the IC miss ratio (the number of IBS IC misses divided by the number of IBS attempted fetches) and the average fetch latency. Fetch latency is the number of cycles from when a fetch is initiated to when the fetch is either completed or aborted. (An **aborted** fetch is a fetch operation that does not complete and deliver instruction data to the decoder.)



2. Select **IBS All ops** from the drop-down list of views.

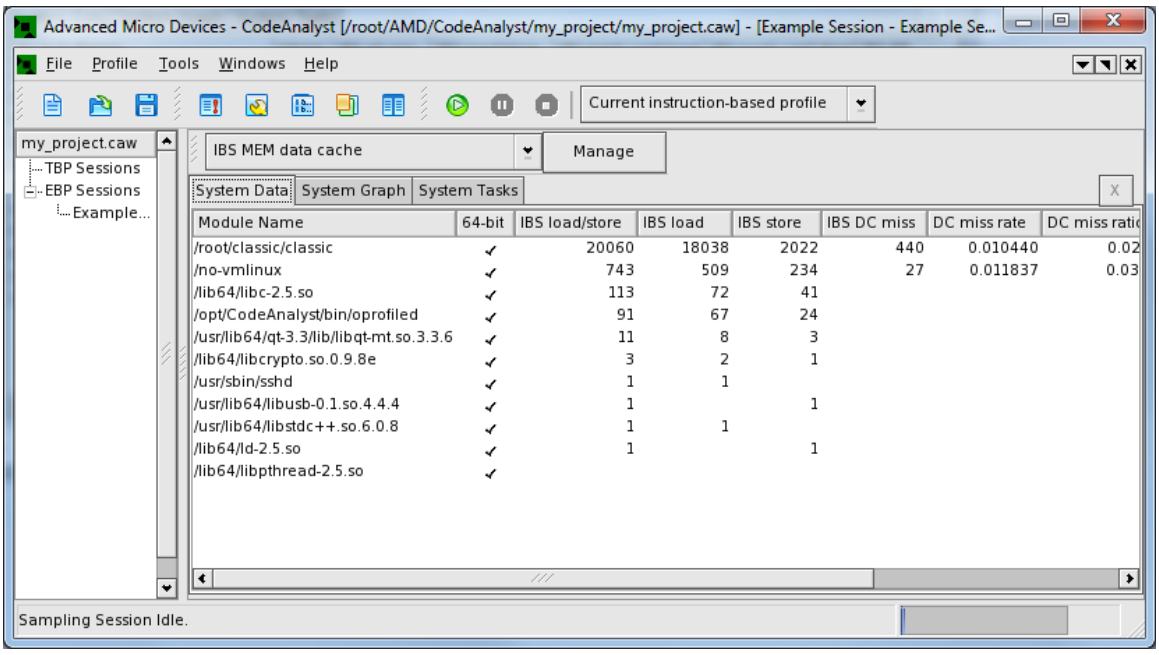
   The "IBS All ops" view displays. This view is an overall summary of the collected IBS op samples. It shows the total number of IBS op samples, the number of op samples taken for branch operations, and the number of samples for ops that performed a memory load and/or store operation. Tag-to-retire time is the number of cycles from when an op was selected (tagged) for sampling to when the op retired. Completion-to-retire time is the number of cycles from when an op completed (finished execution) to when the op retired. Total and average tag-to-retire and completion-to-retire times are shown in the next view.



3. Select the "IBS MEM data cache" view from the drop-down list of views.

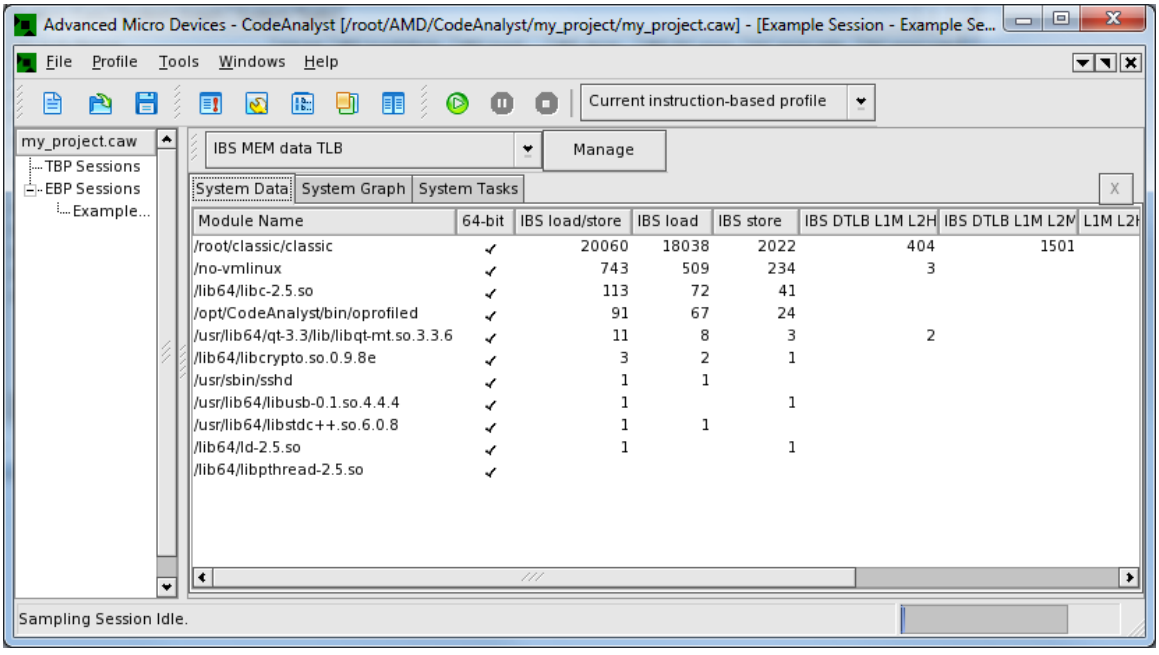
   The "IBS MEM data cache" view is displayed. This view shows information related to data cache (DC) behavior. The number of sampled IBS load/store operations is shown along with a breakdown of the number of loads and the number of stores. The number of IBS samples where the load/

store operation missed in the data cache is shown. The DC miss rate (DC misses divided by the total number of op samples) and DC miss ratio (DC misses divided by the number of load/store operations) are also displayed.



4. Select the "IBS MEM data TLB" view from the drop-down list of views.

   The "IBS MEM data TLB" view is displayed. This view shows information related to data translation lookaside buffer (DTLB) behavior. The number of sampled IBS load/store operations is shown along with a breakdown of the number of load operations and the number of store operations. AMD processors use a two-level DTLB. Address translation may hit in the L1 DTLB, miss in the L1 DTLB and hit in the L2 DTLB ("L1M L2H"), or miss in both levels of the DTLB ("L1M L2M".) The performance penalty for a miss in both levels is relatively high. Nearly half of the sampled load/store operations incurred a missed at both levels of the DTLB. This is the performance culprit in the sample program, classic, which performs a "textbook" implementation of matrix multiplication.

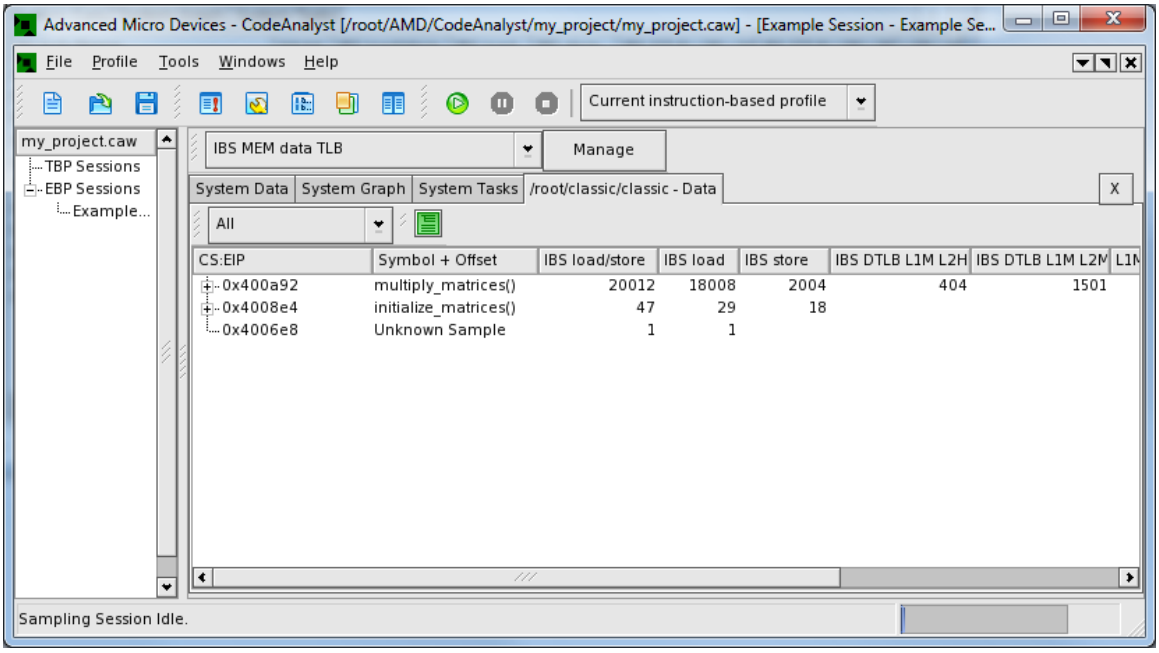


8.6.3. Drilling Down Into IBS Data

In order to find the source of the performance issue in the example program, we need to drill down into the classic module.

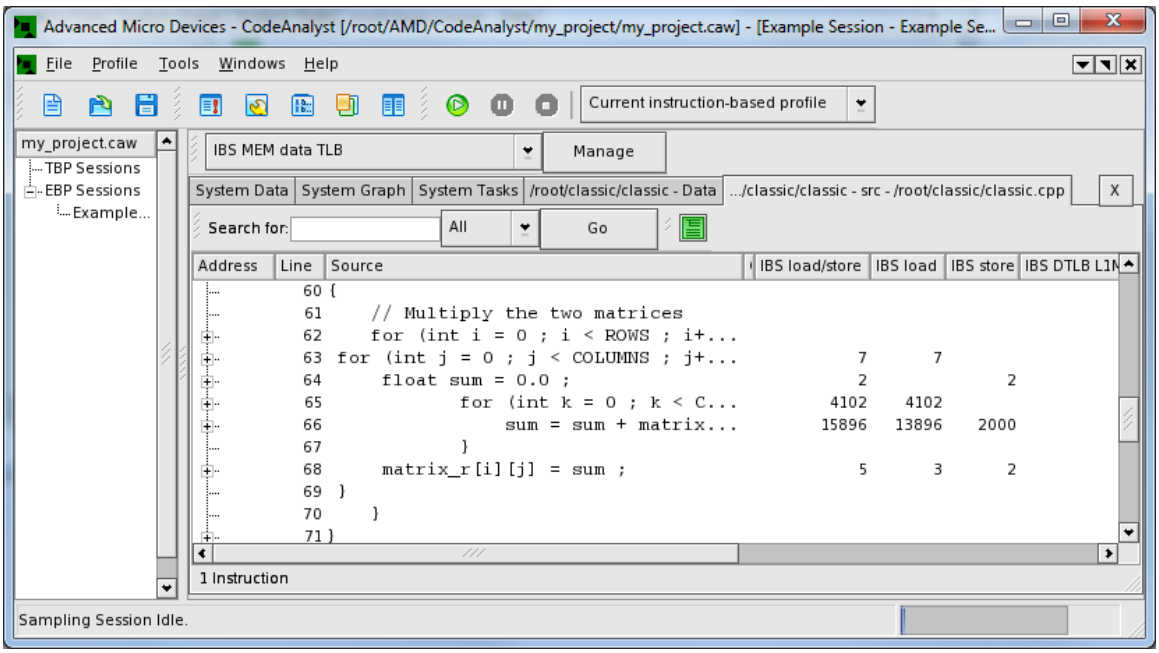
1. Double-click on the module name **classic** in the System Data table.

   A list of functions within classic is displayed with the IBS information for each function. CodeAnalyst retains the "IBS MEM data TLB" view. The function "multiply_matrices" has the most load/store activity and incurs the bulk of the DTLB misses.



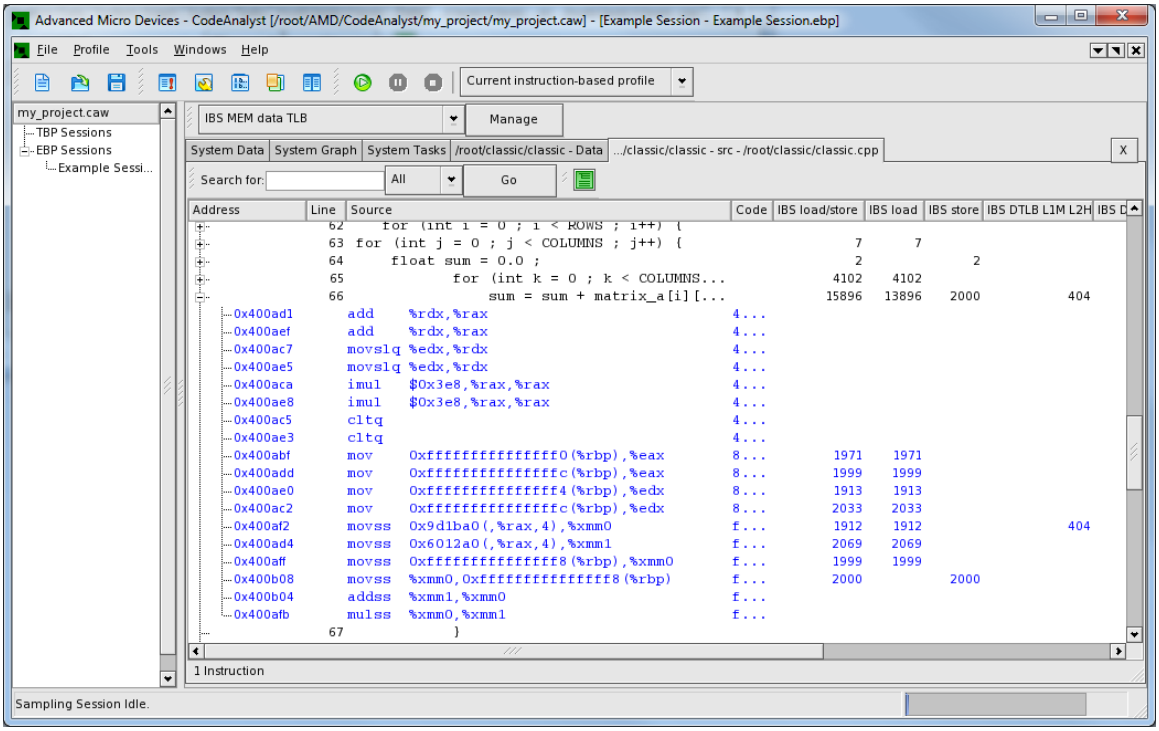
2. Double-click on the function **multiply_matrices** in the Module Data table, i.e., the table of functions within classic.

   The source code for the function "multiply_matrices" is displayed with the IBS information for each source line in the function. Most load/store activity occurs at line 66, which is the statement within the nested loops. This is the statement that reads an element from each of the two operand matrices and computes the running sum of the product of the elements. The DTLB misses are caused by the large strides taken through matrix_b. With nearly every iteration the program touches a different page, thereby causing a miss in the DTLB.



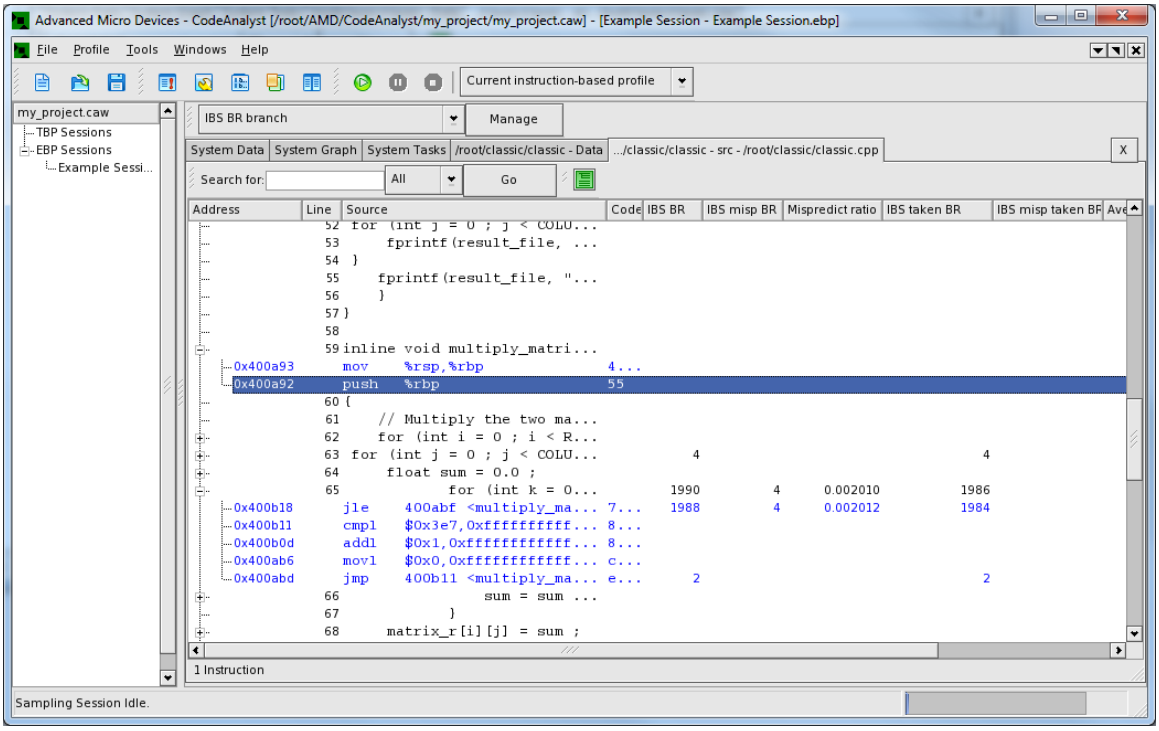
3. Click the expand box "+" to the left of line 66

   The disassembled instructions for source line 66 are displayed along with the IBS data for each instruction. IBS load/store information is attributed to each instruction that performs either a memory read or write operation. Sources of performance-robbing DTLB misses are precisely identified.



4. Select **IBS BR branch** from the drop-down list of views.

   The "IBS BR branch" view displays. This view shows the number of IBS branch op samples and indicates if the branch operation mispredicted and/or was taken. Note that only the conditional jump instruction at the end of the innermost loop is marked as a branch instruction. This example further illustrates the precision offered by Instruction-Based Sampling.



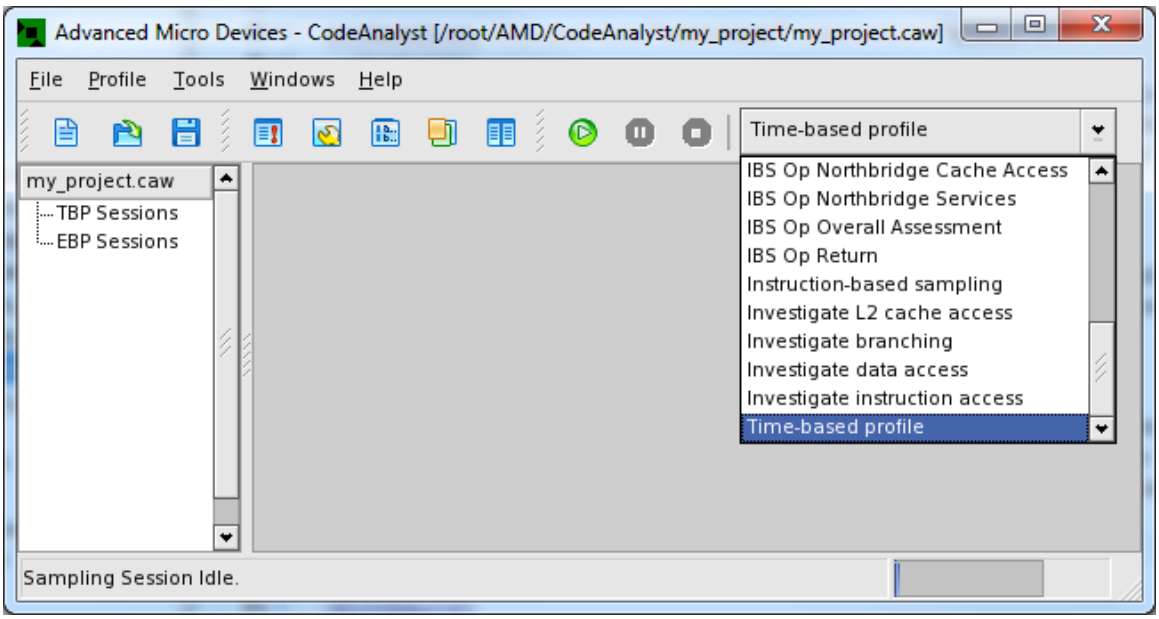
8.7. Tutorial - Profiling a Java Application

This section demonstrates how to analyze a Java program using a time-based profile. The application program used in this section is the SciMark 2.0 benchmark from the National Institute of Standards and Technology (NIST). Source code for the benchmark can be downloaded from http://math.nist.gov/scimark2/scimark2src.zip. You will need a Java compiler in order to compile the benchmark. CodeAnalyst supports both the Sun Microsystems and IBM versions of the Java Virtual Machine (JVM).

Before starting, ensure that a CodeAnalyst project is open and ready for use. The Section 8.3, “Tutorial - Creating a CodeAnalyst Project” demonstrates how to create a new project.

1. Click the **"New"** button in the toolbar or select **"File > New"** from the file menu to create a new project, or click the **"Open"** button in the toolbar to open an existing project or select **"File > Open"** from the file menu. The session settings are slightly different for a Java application.

2. With an open CodeAnalyst project, select **"Time-based profile"** from the drop-down list of profile configurations in the toolbar.



3. Click the **"Session Settings"** button in the toolbar to change the session settings. You may alternately select **"Tools > Session Settings"** from the file menu. A dialog box appears asking for session settings.

4. Change the session name to **"JavaSession"**. Java programs are executed by the JVM, which is started by the Java application launcher tool ("/path-to-java-installation/bin/java").

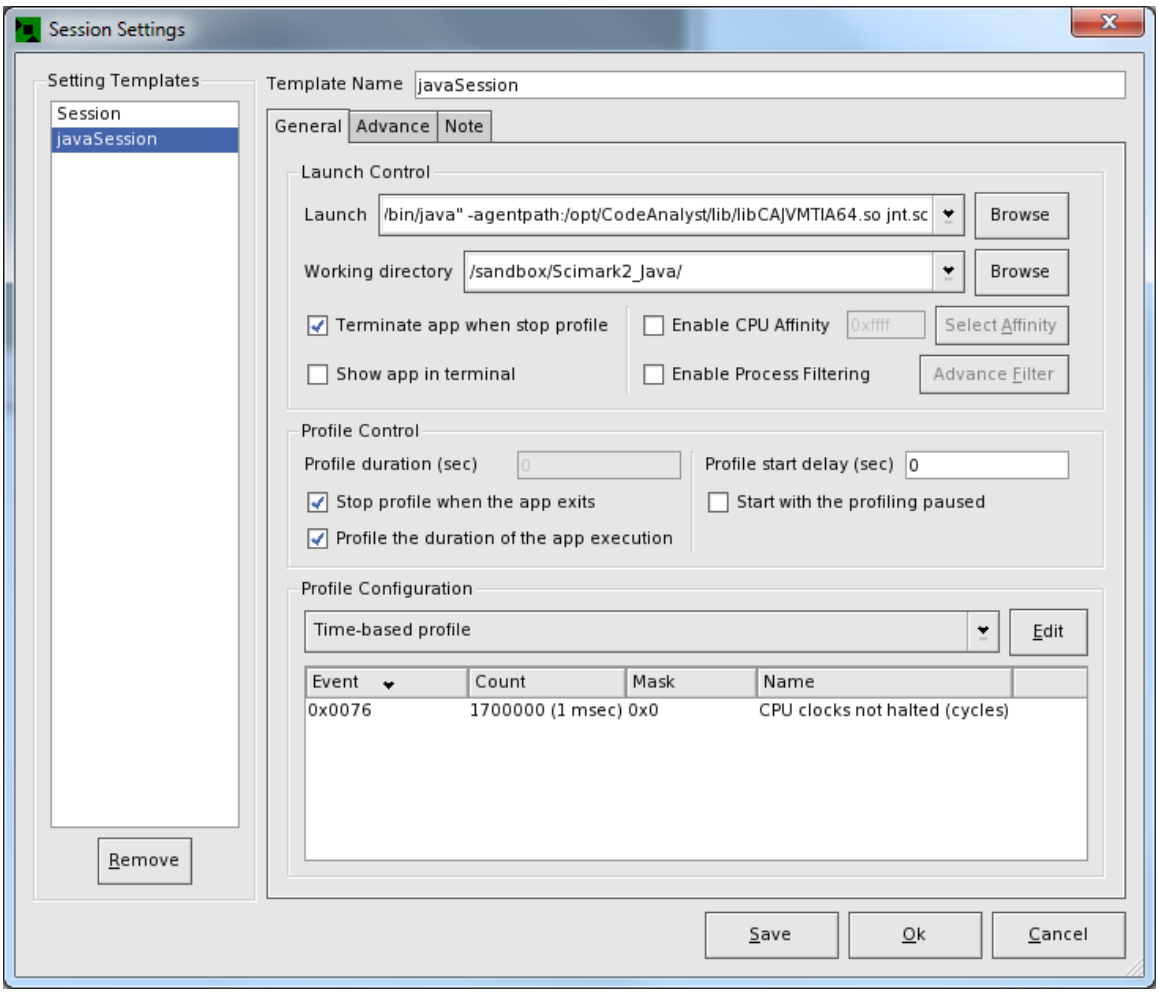
5. In the **"Launch"** field, enter the path `/path-to-java-installation/bin/java` or browse to it by clicking the **"Browse"** button located next to the Launch field.

6. After the path to java, enter `jnt.scimark2.commandline` (the name of the Java application program to be launched).

   Note: CodeAnalyst automatically checks for the Java application launcher tool and inserts the profiling agent library with option **-agentpath** into the Launch field. You do not need to enter this option yourself. The agent connects the JVM to CodeAnalyst.

7. Enter the path to the working directory, or browse to the working directory by clicking the **"Browse"**) button next to the working directory field.

8. Ensure the checkbox is selected to enable **"Terminate the app after the profile"**, **"Stop data collection when the app exits"**, and **"Profile the duration of the app execution"** options. CodeAnalyst collects data for the entire duration of the benchmark run. Click **"OK"** to confirm the session settings and to dismiss the dialog box.

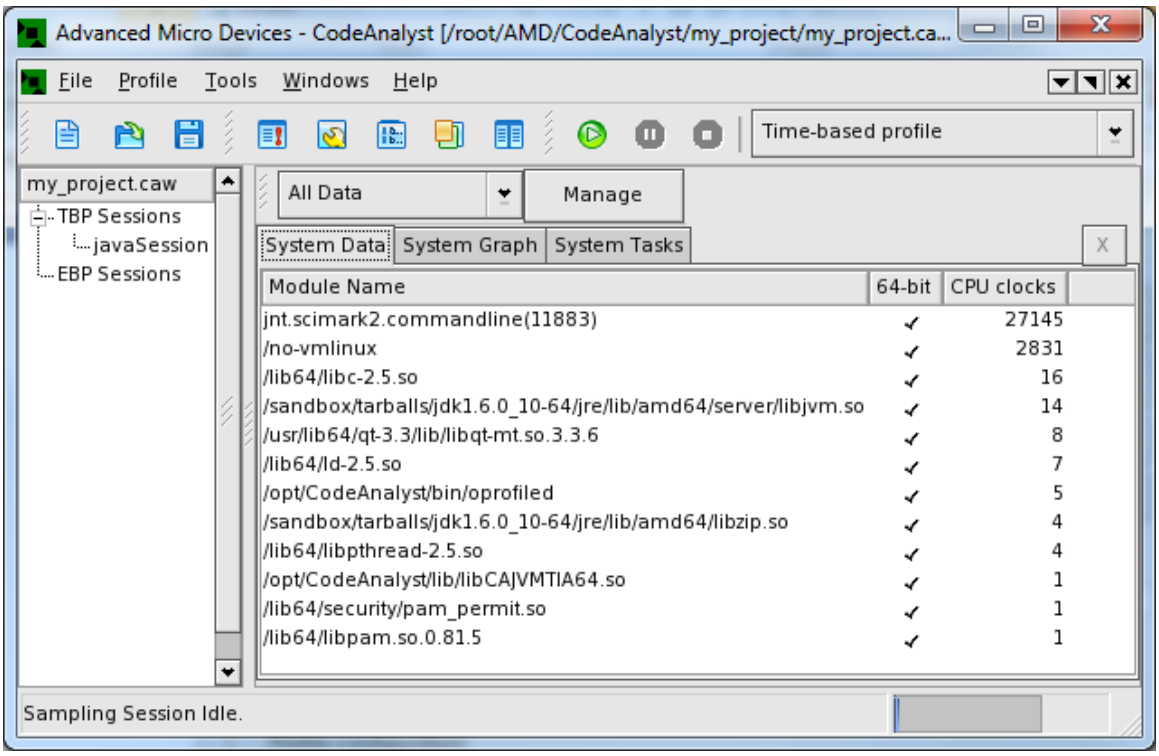


9. Click the **"Start"** button in the toolbar, or select **"Profile > Start"** from the menu.

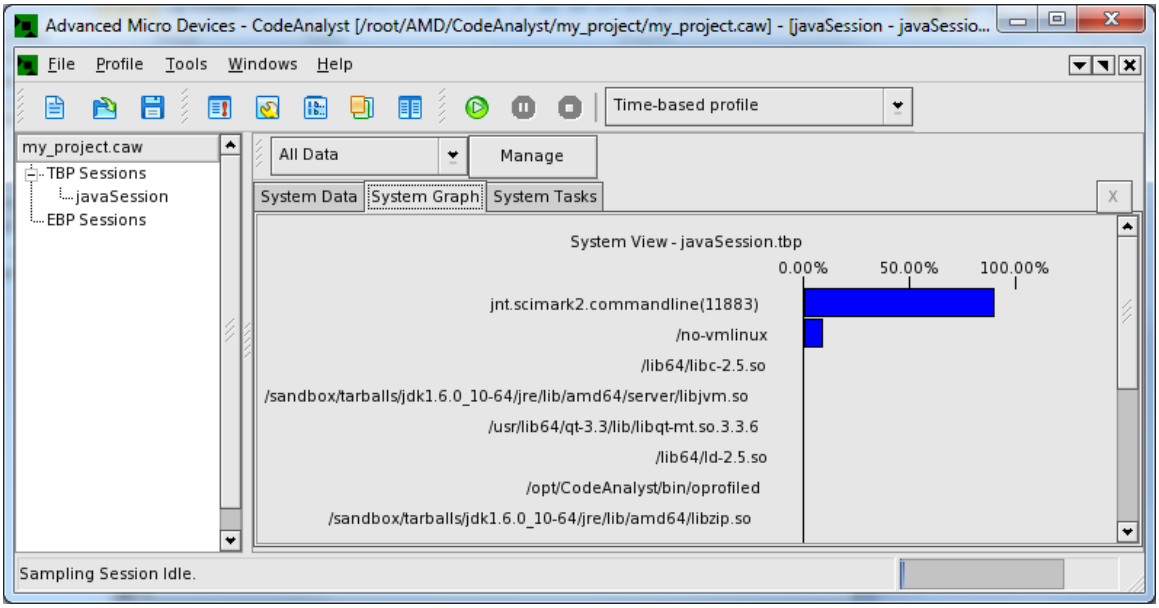
   CodeAnalyst starts data collection and launches the Java application program through the Java application launcher tool. When the benchmark program terminates and data collection is finished, CodeAnalyst displays results in three tabbed panels System Data, System Graph and Processes. The System Data table shows a module-by-module breakdown of timer samples. Each timer sample represents approximately 1 millisecond of execution time (when using the default timer interval of 1 millisecond).

8.7.1. Reviewing Results

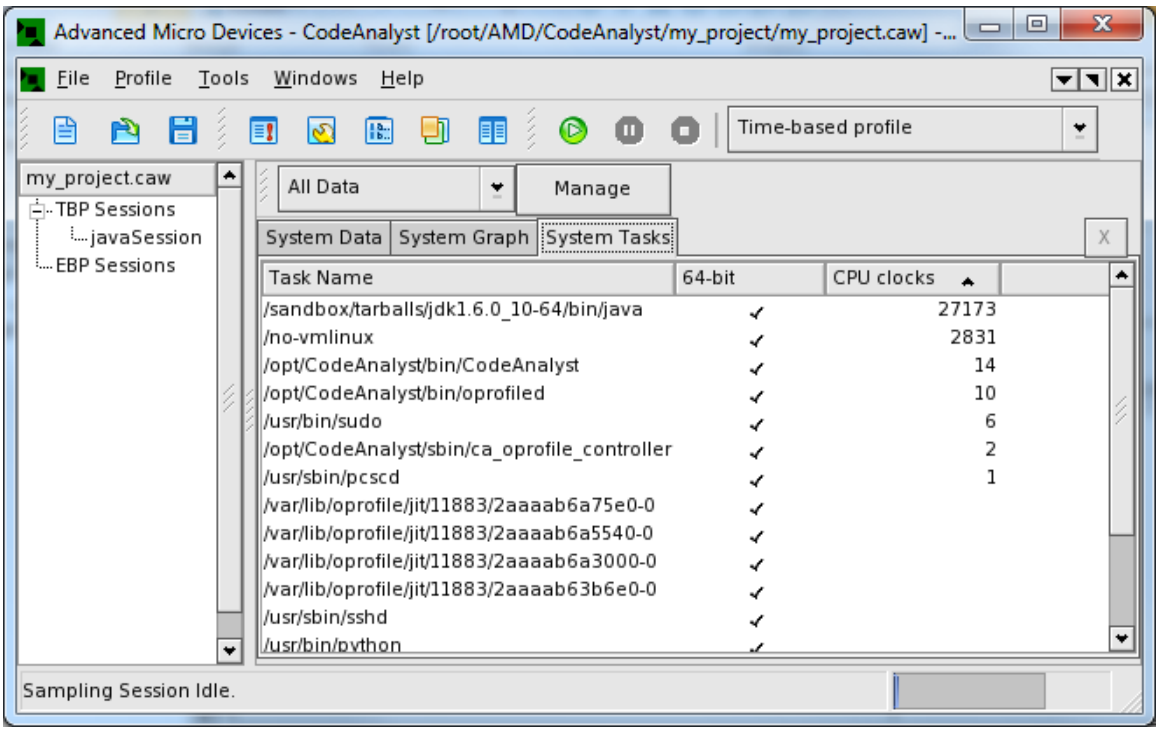
The target java application (**jnt.scimark2.commandline(11883)**)is shown along with the tgid in the "Module Name" column.



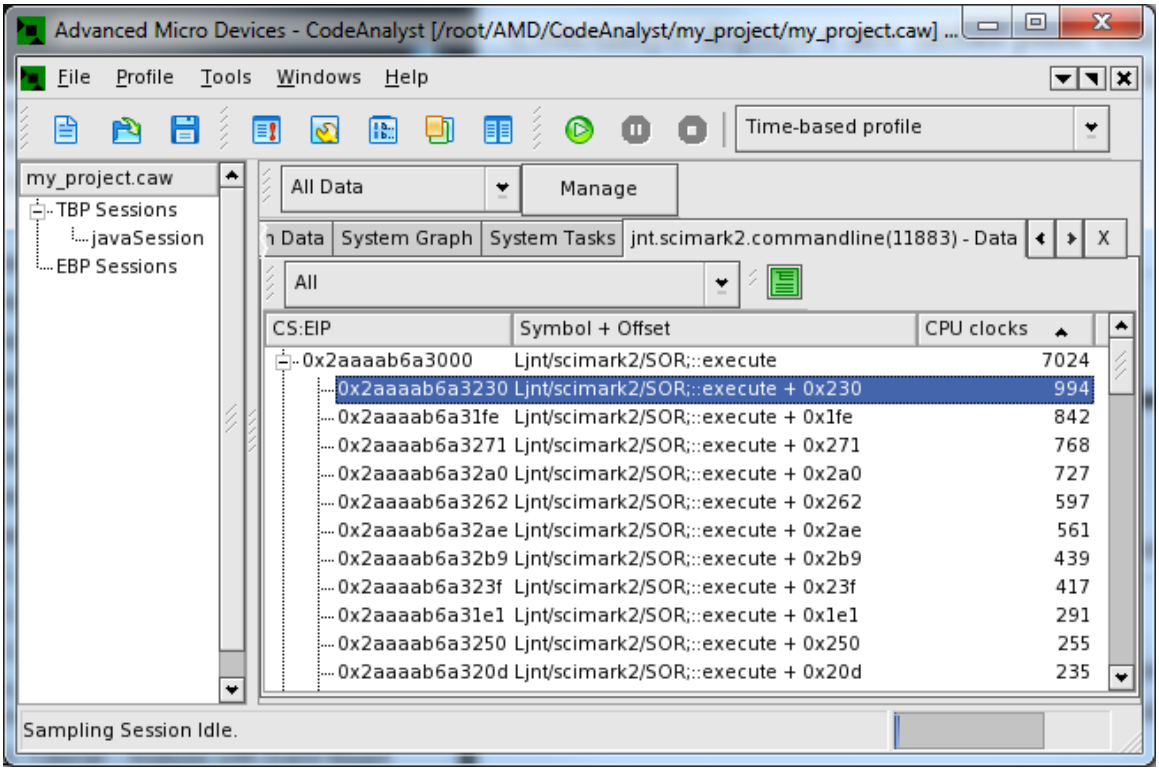
The System Graph shows the module-by-module breakdown of timer samples in the form of a bar chart.



The Processes table displays the distribution of timer samples across the software processes that were active during data collection. CodeAnalyst monitors activity on a system-wide basis. It collects data on any active software component application programs, libraries, device drivers or kernel modules. Time spent idle is ascribed to the system-idle process.



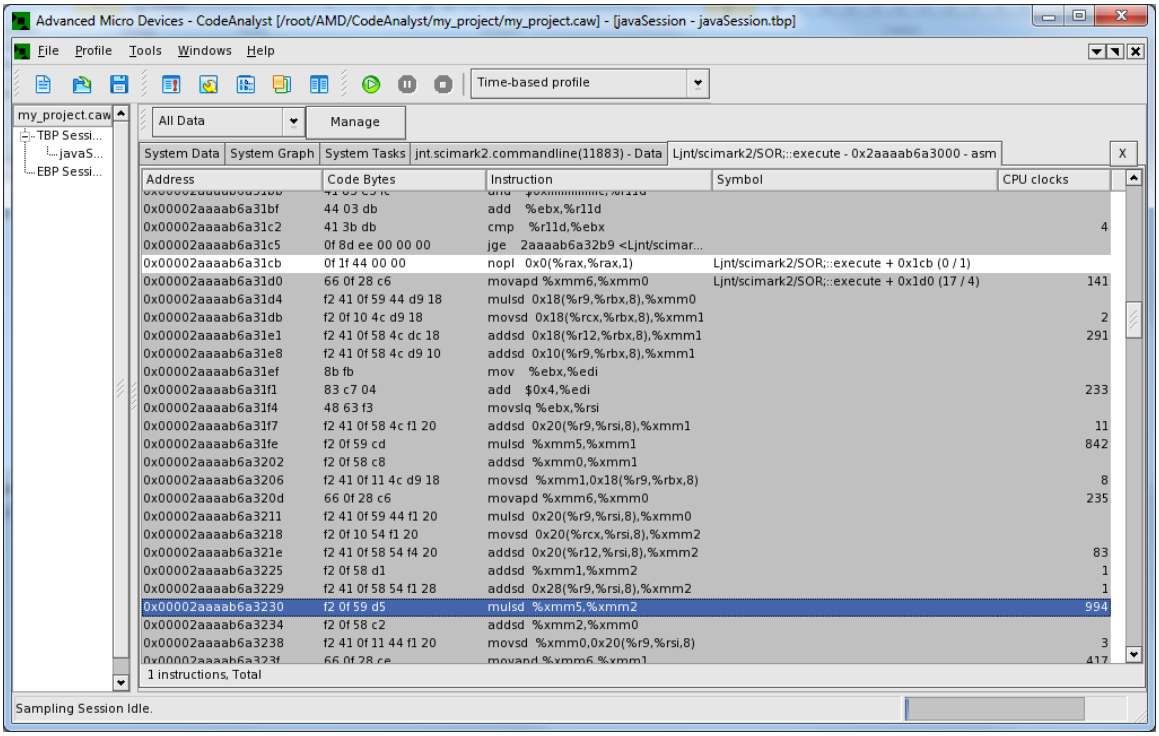
1. Double-click on the module **jnt.scimark2.commandline** to drill down into the performance data for the benchmark program. CodeAnalyst displays a new tabbed panel containing a function-by-function breakdown of timer samples. Use this table to identify functions that consume the most execution time. The hottest functions are the best candidates for optimization.



2. Expand the line **"jnt/scimark2/SOR::execute"**, and double-click on the offset with the highest number of samples (0x00002aaaab6a3230 in this example) to drill down into the function.

CodeAnalyst displays a new tabbed panel containing the annotated assembly code for the function **SOR::execute**. The number of timer samples for each instruction is shown.

Note: CodeAnalyst Linux® does not provide source view for Java application. CodeAnalyst monitors the execution of just-in-time (JIT) compiled code.



8.7.2. Launching a Java Program from the Command Line

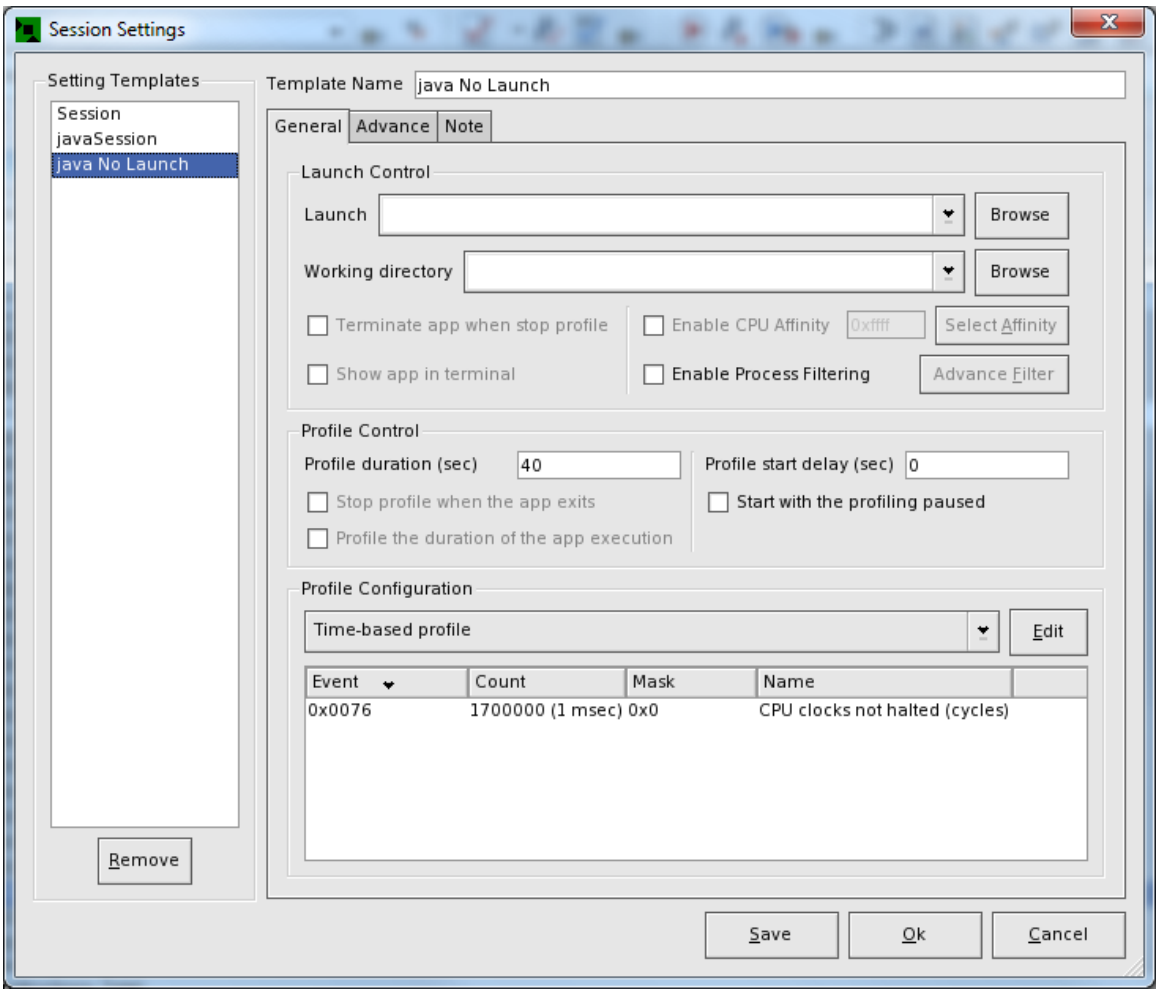
You may also use CodeAnalyst to collect data on a Java program (or any other application program) that is launched from a command line. This is sometimes more convenient than launching the application from within CodeAnalyst.

1. Click the **"Session Settings"** button in the toolbar to change the session settings or, alternately, select **"Tools > Session Settings"** from the menu. A dialog box appears asking for changes to the session settings.

2. Leave the **"Launch"** and **"Working directory"** fields empty. CodeAnalyst disables the Stop data collection when the app exits and Profile the duration of the app execution options.

3. Change the Profile duration field to **40 seconds**.

4. Click **OK** to confirm the changes and to dismiss the dialog box.



5. Click the **Start** button in the toolbar or select **"Profile > Start"** from the menu. CodeAnalyst starts to collect data and will do so for the next 40 seconds as specified by the Profile duration field in the session settings dialog box.

6. At the shell command line, enter the following command to start the benchmark program:

   java -agentpath:/opt/CodeAnalyst//bin/libCAJVMTIA64.so jnt.scimark2.commandline

   You must use the CodeAnalyst profiling agent that is appropriate for the JVM. After 40 seconds has expired, CodeAnalyst stops data collection and displays results in its workspace.

   Some versions of the JVM have deprecated the use of the **-XrunCAJVMPIA** option. Java **-X** options are non-standard and are subject to change without notice.

Chapter 9. Performance Monitoring Events

9.1. Performance Monitoring Events (PME)

AMD processors provide many performance monitoring events to help analyze the performance of programs. The performance monitoring events available for use depend upon the underlying AMD processor which executes the program or system under analysis. Each processor family (and possibly revision within a family) offers a specific range of performance monitoring events. Use the links below to browse the performance monitoring events for each specific processor family.

- BIOS and Kernel Developer's Guide (BKDG) For AMD Family 11h Processor [http://support.amd.com/us/Processor_TechDocs/41256.pdf]
- BIOS and Kernel Developer's Guide (BKDG) For AMD Family 10h Processor [http://support.amd.com/us/Processor_TechDocs/31116.pdf]
- BIOS and Kernel Developer's Guide for AMD Athlon™ and AMD Opteron™ Processors (Rev. A-E) http://support.amd.com/us/Processor_TechDocs/26094.PDF
- BIOS and Kernel Developer's Guide for AMD NPT Family 0Fh Processors (Rev. F-G) http://support.amd.com/us/Processor_TechDocs/32559.pdf
- Section 9.3, “Instruction-Based Sampling Derived Events”

9.2. Unit masks for PMEs

The following three tables show how to construct a unit mask and express combinations of unit masks for a given event.

Unit Mask [bits 15:8]	Mask as Shown in Event Table (bit position in mask)	Unit Mask Hex Value in Event:Unit Mask
xxxx_xxx1	0	0x0001
xxxx_xx1x	1	0x0002
xxxx_x1xx	2	0x0004
xxxx_1xxx	3	0x0008
xxx1_xxxx	4	0x0010
xx1x_xxxx	5	0x0020
x1xx_xxxx	6	0x0040
Reserved		

Unit Mask [bits 15:8]	Combined Masks (combined bit positions)	Unit Mask Hex Value in Event: Unit Mask
xxxx_xx11	0 and 1	0x0003
xxxx_x1x1	0 and 2	0x0005
xxxx_x11x	1 and 2	0x0006
xxxx_x111	0 and 1 and 2	0x0007
xxx1_xx1x	1 and 4	0x0012

Unit Mask [bits 15:8]	Combined Masks (combined bit positions)	Unit Mask Hex Value in Event: Unit Mask
xxx1_x1x1	0 and 2 and 4	0x0015
xxx1_1111	0 and 1 and 2 and 3 and 4	0x001F

Event	Unit Mask	Description	Event: Unit Mask
0x41		Data cache refill from L2	
	0	**I**nvalid cache state	0x4101
	1	**S**hared cache state	0x4102
	2	**E**xclusive cache state	0x4104
	3	**O**wner cache state	0x4108
	4	**M**odified cache state	0x4110
	All of the above	All **MOESI** states	0x411F
	0 and 1 and 3	**I**nvalid and **S**hared and **O**wner states	0x410B

9.3. Instruction-Based Sampling Derived Events

This section describes the derived events for Instruction-Based Sampling (IBS.) IBS is available on AMD Family 10h processors.

9.3.1. IBS Fetch Derived Events

9.3.1.1. Event 0xF000

Abbreviation: IBS fetch

The number of all IBS fetch samples. This derived event counts the number of all IBS fetch samples that were collected including IBS-killed fetch samples.

9.3.1.2. Event 0xF001

Abbreviation: IBS fetch killed

The number of IBS sampled fetches that were killed fetches. A fetch operation is *killed* if the fetch did not reach ITLB or IC access. The number of killed fetch samples is not generally useful for analysis and are filtered out in other derived IBS fetch events (except Event Select 0xF000 which counts all IBS fetch samples including IBS killed fetch samples.)

9.3.1.3. Event 0xF002

Abbreviation: IBS fetch attempt

The number of IBS sampled fetches that were not killed fetch attempts. This derived event measures the number of useful fetch attempts and does not include the number of IBS killed fetch samples. This event should be used to compute ratios such as the ratio of IBS fetch IC misses to attempted fetches.

The number of attempted fetches should equal the sum of the number of completed fetches and the number of aborted fetches.

9.3.1.4. Event 0xF003

Abbreviation: IBS fetch comp

The number of IBS sampled fetches that completed. A fetch is *completed* if the attempted fetch delivers instruction data to the instruction decoder. Although the instruction data was delivered, it may still not be used (e.g., the instruction data may have been on the "wrong path" of an incorrectly predicted branch.)

9.3.1.5. Event 0xF004

Abbreviation: IBS fetch abort

The number of IBS sampled fetches that aborted. An attempted fetch is *aborted* if it did not complete and deliver instruction data to the decoder. An attempted fetch may abort at any point in the process of fetching instruction data. An abort may be due to a branch redirection as the result of a mispredicted branch.

The number of IBS aborted fetch samples is a lower bound on the amount of unsuccessful, speculative fetch activity. It is a lower bound since the instruction data delivered by completed fetches may not be used.

9.3.1.6. Event 0xF005

Abbreviation: IBS L1 ITLB hit

The number of IBS attempted fetch samples where the fetch operation initially hit in the L1 ITLB (Instruction Translation Lookaside Buffer).

9.3.1.7. Event 0xF006

Abbreviation: IBS ITLB L1M L2H

The number of IBS attempted fetch samples where the fetch operation initially missed in the L1 ITLB and hit in the L2 ITLB.

9.3.1.8. Event 0xF007

Abbreviation: IBS ITLB L1M L2M

The number of IBS attempted fetch samples where the fetch operation initially missed in both the L1 ITLB and the L2 ITLB.

9.3.1.9. Event 0xF008

Abbreviation: IBS IC miss

The number of IBS attempted fetch samples where the fetch operation initially missed in the IC (instruction cache).

9.3.1.10. Event 0xF009

Abbreviation: IBS IC hit

The number of IBS attempted fetch samples where the fetch operation initially hit in the IC.

9.3.1.11. Event 0xF00A

Abbreviation: IBS 4K page

The number of IBS attempted fetch samples where the fetch operation produced a valid physical address (i.e., address translation completed successfully) and used a 4-KByte page entry in the L1 ITLB.

9.3.1.12. Event 0xF00B

Abbreviation: IBS 2M page

The number of IBS attempted fetch samples where the fetch operation produced a valid physical address (i.e., address translation completed successfully) and used a 2-MByte page entry in the L1 ITLB.

9.3.1.13. Event 0xF00E

Abbreviation: IBS fetch lat

The total latency of all IBS attempted fetch samples. Divide the total IBS fetch latency by the number of IBS attempted fetch samples to obtain the average latency of the attempted fetches that were sampled.

9.3.2. IBS Op Derived Events

9.3.2.1. Event 0xF100

Abbreviation: IBS all ops

The number of all IBS op samples that were collected. These op samples may be branch ops, resync ops, ops that perform load/store operations, or undifferentiated ops (e.g., those ops that perform arithmetic operations, logical operations, etc.).

IBS collects data for retired ops. No data is collected for ops that are aborted due to pipeline flushes, etc. Thus, all sampled ops are architecturally significant and contribute to the successful forward progress of executing programs.

9.3.2.2. Event 0xF101

Abbreviation: IBS tag-to-ret

The total number of tag-to-retire cycles across all IBS op samples. The tag-to-retire time of an op is the number of cycles from when the op was tagged (selected for sampling) to when the op retired.

9.3.2.3. Event 0xF102

Abbreviation: IBS comp-to-ret

The total number of completion-to-retire cycles across all IBS op samples. The completion-to-retire time of an op is the number of cycles from when the op completed to when the op retired.

9.3.3. IBS Op Branches Derives Events

9.3.3.1. Event 0xF103

Abbreviation: IBS BR

The number of IBS retired branch op samples. A branch operation is a change in program control flow and includes unconditional and conditional branches, subroutine calls and subroutine returns. Branch ops are used to implement AMD64 branch semantics.

9.3.3.2. Event 0xF104

Abbreviation: IBS misp BR

The number of IBS samples for retired branch operations that were mispredicted. This event should be used to compute the ratio of mispredicted branch operations to all branch operations.

9.3.3.3. Event 0xF105

Abbreviation: IBS taken BR

The number of IBS samples for retired branch operations that were taken branches.

9.3.3.4. Event 0xF106

Abbreviation: IBS misp taken BR

The number of IBS samples for retired branch operations that were mispredicted taken branches.

9.3.3.5. Event 0xF107

Abbreviation: IBS RET

The number of IBS retired branch op samples where the operation was a subroutine return. These samples are a subset of all IBS retired branch op samples.

9.3.3.6. Event 0xF108

Abbreviation: IBS misp RET

The number of IBS retired branch op samples where the operation was a mispredicted subroutine return. This event should be used to compute the ratio of mispredicted returns to all subroutine returns.

9.3.3.7. Event 0xF109

Abbreviation: IBS resync

The number of IBS resync op samples. A resync op is only found in certain microcoded AMD64 instructions and causes a complete pipeline flush.

9.3.4. IBS Op Load-Store Derived Events

9.3.4.1. Event 0xF200

Abbreviation: IBS load/store

The number of IBS op samples for ops that perform either a load and/or store operation.

An AMD64 instruction may be translated into one ("single fastpath"), two ("double fastpath"), or several ("vector path") ops. Each op may perform a load operation, a store operation or both a load and store operation (each to the same address). Some op samples attributed to an AMD64 instruction may perform a load/store operation while other op samples attributed to the same instruction may not. Further, some branch instructions perform load/store operations. Thus, a mix of op sample types may be attributed to a single AMD64 instruction depending upon the ops that are issued from the AMD64 instruction and the op types.

9.3.4.2. Event 0xF201

Abbreviation: IBS load

The number of IBS op samples for ops that perform a load operation.

9.3.4.3. Event 0xF202

Abbreviation: IBS store

The number of IBS op samples for ops that perform a store operation.

9.3.4.4. Event 0xF203

Abbreviation: IBS L1 DTLB hit

The number of IBS op samples where either a load or store operation initially hit in the L1 DTLB (data translation lookaside buffer).

9.3.4.5. Event 0xF204

Abbreviation: IBS DTLB L1M L2H

The number of IBS op samples where either a load or store operation initially missed in the L1 DTLB and hit in the L2 DTLB.

9.3.4.6. Event 0xF205

Abbreviation: IBS DTLB L1M L2M

The number of IBS op samples where either a load or store operation initially missed in both the L1 DTLB and the L2 DTLB.

9.3.4.7. Event 0xF206

Abbreviation: IBS DC miss

The number of IBS op samples where either a load or store operation initially missed in the data cache (DC).

9.3.4.8. Event 0xF207

Abbreviation: IBS DC hit

The number of IBS op samples where either a load or store operation initially hit in the data cache (DC).

9.3.4.9. Event 0xF208

Abbreviation: IBS misalign acc

The number of IBS op samples where either a load or store operation caused a misaligned access (i.e., the load or store operation crossed a 128-bit boundary).

9.3.4.10. Event 0xF209

Abbreviation: IBS bank conf load

The number of IBS op samples where either a load or store operation caused a bank conflict with a load operation.

9.3.4.11. Event 0xF20A

Abbreviation: IBS bank conf store

The number of IBS op samples where either a load or store operation caused a bank conflict with a store operation.

9.3.4.12. Event 0xF20B

Abbreviation: IBS forwarded

The number of IBS op samples where data for a load operation was forwarded from a store operation.

9.3.4.13. Event 0xF20C

Abbreviation: IBS cancelled

The number of IBS op samples where data forwarding to a load operation from a store was cancelled.

9.3.4.14. Event 0xF20D

Abbreviation: IBS UC mem acc

The number of IBS op samples where a load or store operation accessed uncacheable (UC) memory.

9.3.4.15. Event 0xF20E

Abbreviation: IBS WC mem acc

The number of IBS op samples where a load or store operation accessed write combining (WC) memory.

9.3.4.16. Event 0xF20F

Abbreviation: IBS locked op

The number of IBS op samples where a load or store operation was a locked operation.

9.3.4.17. Event 0xF210

Abbreviation: IBS MAB hit

The number of IBS op samples where a load or store operation hit an already allocated entry in the Miss Address Buffer (MAB).

9.3.4.18. Event 0xF211

Abbreviation: IBS L1 DTLB 4K

The number of IBS op samples where a load or store operation produced a valid linear (virtual) address and a 4-KByte page entry in the L1 DTLB was used for address translation.

9.3.4.19. Event 0xF212

Abbreviation: IBS L1 DTLB 2M

The number of IBS op samples where a load or store operation produced a valid linear (virtual) address and a 2-MByte page entry in the L1 DTLB was used for address translation.

9.3.4.20. Event 0xF213

Abbreviation: IBS L1 DTLB 1G

The number of IBS op samples where a load or store operation produced a valid linear (virtual) address and a 1-GByte page entry in the L1 DTLB was used for address translation.

9.3.4.21. Event 0xF215

Abbreviation: IBS L2 DTLB 4K

The number of IBS op samples where a load or store operation produced a valid linear (virtual) address, hit the L2 DTLB, and used a 4 KByte page entry for address translation.

9.3.4.22. Event 0xF216

Abbreviation: IBS L2 DTLB 2M

The number of IBS op samples where a load or store operation produced a valid linear (virtual) address, hit the L2 DTLB, and used a 2-MByte page entry for address translation.

9.3.4.23. Event 0xF219

Abbreviation: IBS DC load lat

The total DC miss latency (in processor cycles) across all IBS op samples that performed a load operation. The miss latency is the number of clock cycles from when the data cache miss was detected to when data was delivered to the core. Divide the total DC miss latency by the number of sampled load operations to obtain the average DC miss latency.

9.3.5. IBS Op Northbridge Derived Events

9.3.5.1. Event 0xF240

Abbreviation: IBS NB local

The number of IBS op samples where a load operation was serviced from the local processor.

Northbridge IBS data is only valid for load operations that miss in both the L1 data cache and the L2 data cache. If a load operation crosses a cache line boundary, then the IBS data reflects the access to the lower cache line.

9.3.5.2. Event 0xF241

Abbreviation: IBS NB remote

The number of IBS op samples where a load operation was serviced from a remote processor.

9.3.5.3. Event 0xF242

Abbreviation: IBS NB local L3

The number of IBS op samples where a load operation was serviced by the local L3 cache.

9.3.5.4. Event 0xF243

Abbreviation: IBS NB local cache

The number of IBS op samples where a load operation was serviced by a cache (L1 data cache or L2 cache) belonging to a local core which is a sibling of the core making the memory request.

9.3.5.5. Event 0xF244

Abbreviation: IBS NB remote cache

The number of IBS op samples where a load operation was serviced by a remote L1 data cache, L2 cache or L3 cache after traversing one or more coherent HyperTransport™ links.

9.3.5.6. Event 0xF245

Abbreviation: IBS NB local DRAM

The number of IBS op samples where a load operation was serviced by local system memory (local DRAM via the memory controller).

9.3.5.7. Event 0xF246

Abbreviation: IBS NB remote DRAM

The number of IBS op samples where a load operation was serviced by remote system memory (after traversing one or more coherent HyperTransport links and through a remote memory controller).

9.3.5.8. Event 0xF247

Abbreviation: IBS NB local other

The number of IBS op samples where a load operation was serviced from local MMIO, configuration or PCI space, or from the local APIC.

9.3.5.9. Event 0xF248

Abbreviation: IBS NB remote other

The number of IBS op samples where a load operation was serviced from remote MMIO, configuration or PCI space.

9.3.5.10. Event 0xF249

Abbreviation: IBS NB cache M

The number of IBS op samples where a load operation was serviced from local or remote cache, and the cache hit state was the Modified (M) state.

9.3.5.11. Event 0xF24A

Abbreviation: IBS NB cache O

The number of IBS op samples where a load operation was serviced from local or remote cache, and the cache hit state was the Owned (O) state.

9.3.5.12. Event 0xF24B

Abbreviation: IBS NB local lat

The total data cache miss latency (in processor cycles) for load operations that were serviced by the local processor.

9.3.5.13. Event 0xF24C

Abbreviation: IBS NB remote lat

The total data cache miss latency (in processor cycles) for load operations that were serviced by a remote processor.

Chapter 10. Support

10.1. Enhancement Request

Please email the following information about a desired enhancement or change to CodeAnalyst.support@amd.com [mailto:CodeAnalyst.support@amd.com]:

- State which version of AMD CodeAnalyst you are using. Choose **Help > About** to view the About AMD CodeAnalyst dialog box.
- Describe the desired enhancement or change.
- Indicate to us how important this is to you using a scale of 1 to 5 where 1 is most important and 5 least important.

10.2. Problem Report

If a problem is found, take the following action:

1. Run **careport.sh** script which is located in CodeAnalyst root directory of the source tree, or **/opt/CodeAnalyst/bin/careport.sh**. This script will generate a report file called **CAReport.txt**.
2. Please provide the following information:
   - Give a description of the problem or issue.
   - Briefly describe the steps or sequence of events leading to the observation.
   - State how frequently problem occurred.
   - Describe the messages AMD CodeAnalyst displayed.
   - State which version of the AMD CodeAnalyst was used (under **Help > System Info** or **opcontrol --version**).
   - Describe the application analyzed.
3. Please send the report file (**CAReport.txt**) in step 1 and information in step 2 to CodeAnalyst.support@amd.com [mailto:CodeAnalyst.support@amd.com].

Appendix A. GNU General Public License

Free Software Foundation, Inc.
51 Franklin Street, Fifth Floor,
Boston, MA 02110-1301
USA

Version 2, June 1991

A.1. Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software - to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps:

1. copyright the software, and

2. offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

A.2. TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

A.2.1. Section 0

This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The “Program”, below, refers to any such program or work, and a “work based on the Program” means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term “modification”.) Each licensee is addressed as “you”.

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

A.2.2. Section 1

You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

A.2.3. Section 2

You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section A.2.2, “Section 1” above, provided that you also meet all of these conditions:

a. You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

b. You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

c. If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: If the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

A.2.4. Section 3

You may copy and distribute the Program (or a work based on it, under Section A.2.3, “Section 2” in object code or executable form under the terms of Section A.2.2, “Section 1” and Section A.2.3, “Section 2” above provided that you also do one of the following:

a. Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

b. Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

c. Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

A.2.5. Section 4

You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

A.2.6. Section 5

You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

A.2.7. Section 6

Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

A.2.8. Section 7

If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

A.2.9. Section 8

If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

A.2.10. Section 9

The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and “any later version”, you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

A.2.11. Section 10

If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free

Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

A.2.12. NO WARRANTY Section 11

BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

A.2.13. Section 12

IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

A.3. How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the “copyright” line and a pointer to where the full notice is found.

<one line to give the program's name and a brief idea of what it does.> Copyright (C) <year> <name of author>

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

Gnomovision version 69, Copyright (C) year name of author Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type “show w”. This is free software, and you are welcome to redistribute it under certain conditions; type “show c” for details.

The hypothetical commands “show w” and “show c” should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than “show w” and “show c”; they could even be mouse-clicks or menu items--whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a “copyright disclaimer” for the program, if necessary. Here is a sample; alter the names:

Yoyodyne, Inc., hereby disclaims all copyright interest in the program “Gnomovision” (which makes passes at compilers) written by James Hacker.

<signature of Ty Coon>, 1 April 1989 Ty Coon, President of Vice

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

Appendix B. Features List

B.1. New Features in CodeAnalyst 2.9

- Redesign "Session Setting" dialog
- Redesign "Event Selection" dialog
- Improve JAVA profile to handle JIT modules unloading
- Improve security and permission management for non-root users
- OProfile daemon/driver monitoring tool
- New CA-OProfile based on OProfile-0.9.6

B.2. New Features in CodeAnalyst 2.8

- Support for AMD processor Family 10h Revision 4
- New mode for IBS OP
- Section 3.5, “Basic Block Analysis”
- Section 3.6, “In-Line Analysis”
- True event multiplexing
- Section 3.7, “Session Diff Analysis”
- Session Import by TBP/EBP file

B.3. New Features in CodeAnalyst 2.7

CodeAnalyst 2.7 added support for **Instruction-Based Sampling** (IBS.) Instruction Sampling is a new performance measurement technique supported by AMD Family 10h processors. IBS has these advantages:

- IBS precisely associates hardware event information with the instructions that cause the events. A data cache miss, for example, is associated with the AMD64 instruction performing the memory read or write operation that caused the miss. Improved precision makes it easier to pinpoint specific performance issues.
- IBS collects a wide range of hardware event information in a single measurement run.
- IBS collects new information such as retire delay and data cache miss latency.

See Section 3.4, “Instruction-Based Sampling Analysis” for more information about IBS. The sections Section 8.6, “Tutorial - Analysis with Instruction-Based Sampling Profile” and Section 5.4, “Collecting an Instruction-Based Sampling Profile” give step-by-step directions for collecting IBS profile data.

CodeAnalyst reports IBS performance information as **derived events**. Derived events are similar to the events reported by Section 3.3, “Event-Based Profiling Analysis”. The IBS-derived events are described in Section 9.3, “Instruction-Based Sampling Derived Events”

CodeAnalyst supports the new performance monitoring events provided by AMD Family 10h processors. See BIOS and Kernel Developer's Guide (BKDG) For AMD Family 10h Processor [http://support.amd.com/us/Processor_TechDocs/31116.pdf] for complete list of performance events.

The online help has been restructured and revised. The new structure follows the CodeAnalyst workflow. See the Chapter 1, *Introduction*.

B.4. New Features in CodeAnalyst 2.6

- Chapter 4, *Configure Profile* that assist the configuration of profile data collection.
- Chapter 7, *View Configuration* that make it easier to review and interpret results.
- Section 2.4, “Event Counter Multiplexing” that extends Section 3.3, “Event-Based Profiling Analysis” and makes it possible to measure more than four events in a single run.

Bibliography

BIOS and Kernel Developer's Guide (BKDG)

BIOS and Kernel Developer's Guide (BKDG) For AMD Family 11h Processor
[http://support.amd.com/us/Processor_TechDocs/41256.pdf]

BIOS and Kernel Developer's Guide (BKDG) For AMD Family 10h Processor
[http://support.amd.com/us/Processor_TechDocs/31116.pdf]

BIOS and Kernel Developer's Guide for AMD Athlon™ and AMD Opteron™ Processors (Rev. A-E)
http://support.amd.com/us/Processor_TechDocs/26094.PDF

BIOS and Kernel Developer's Guide for AMD NPT Family 0Fh Processors (Rev. F-G)
http://support.amd.com/us/Processor_TechDocs/32559.pdf

General Documentation

Basic Performance Measurements for AMD Athlon™ 64, AMD Opteron™ and AMD Phenom™ Processors
http://developer.amd.com/Assets/Basic_Performance_Measurements.pdf

Increased performance with AMD CodeAnalyst software and Instruction-Based Sampling (on Linux)
http://developer.amd.com/Assets/amd_ca_linux_june_2008.pdf

An introduction to analysis and optimization with AMD CodeAnalyst Performance Analyzer
http://developer.amd.com/Assets/Introduction_to_CodeAnalyst.pdf

Improving program performance with AMD CodeAnalyst for Linux®
http://developer.amd.com/assets/Linux_Summit_PJD_2007_v2.pdf

Instruction-Based Sampling: A New Performance Analysis Technique for AMD Family 10h Processors
http://developer.amd.com/assets/AMD_IBS_paper_EN.pdf

Index

W

X