## CPU Profiler

CodeXL's CPU Profiler is used for performance analysis and tuning of applications running on CPU.

The CPU Profiler lets you identify main performance bottlenecks of the profiled application or the entire system.

Features

* Three profile modes:
  + Time-Based Profile (TBP) can identify the “hot-spots” in the profiled applications. (Hot-spots are code areas that use significantly more time compared to other areas in the code.)
  + Event-Based Profile (EBP) can identify CPU and memory related performance issues in the profiled applications.
  + Instruction-Based Sampling (IBS) can record and count the instructions that trigger HW events, as well as calculate various metrics, such as data cache latency.
* Per-Process mode and System-Wide mode profiling:
  + **Per-process mode** profiles a process and its children.
  + **System-wide mode** profiles the entire system.
* Attach to process for profiling.
* User-mode profiling and Kernel-mode profiling (Windows only).
* Native applications profiling (C, C++ and Fortran).
* Profiling C++ inline functions.
* CLR/.NET applications profiling (only on Windows).
* Java applications profiling.
* Call Stack Sampling (CSS) for all profile modes.
* Aggregation of the collected samples at various levels: Process/Modules/Functions/Source and Disassembly.
* HW events counter multiplexing.
* Debugging Data Formats Supported:
  + CodeXL supports symbol information for unmanaged Executables compiled by MS Visual Studio or GCC (under Linux or other Unix-like systems (like Cygwin and MinGW)). That includes the following debugging data formats: PDB, COFF, DWARF, STABS.
  + For managed Executables, CodeXL supports Java and .NET applications’ debug information.
* Time-Based Profile (TBP) and Event-Based Profile (EBP) are supported in guest OS running on VMware Workstation 11.0 or later.
* Time-Based Profile (TBP) and Event-Based Profile (EBP) are supported on Microsoft Hyper-V.
* Time-Based Profile (TBP) is supported on Xen Project hypervisor.
* Time-Based Profile (TBP) is supported on Linux KVM hypervisor.
* Controlling CPU Profiling i.e. pause and resume profiling, from target application to limit profiling scope.
* Aggregated instruction based (IMIX) report generation from CPU Profiler CLI.

Limitations

* CPU Profiler expects the profiled application executable binaries must not be compressed or obfuscated by any software protector tools, e.g. VMProtect.

Using the CPU Profiler

This section explains various key concepts related to CPU Profiling. It consists of the following subsections.

* [**CPU Profile Key Concepts**](#_topic_CPUProfileKeyConcepts)
* [**CPU Profile Configurations**](#_topic_CPUProfileConfigurations)
* [**CPU Profile Session**](#_topic_CPUProfileSession)
* [**CPU Profile Data Analysis**](#_topic_CPUProfileDataAnalysis)
* [**CPU Profile Command Line Interface**](#_CPU_Profile_Command)
* [**CPU Profile C/C++ Inline Functions**](#_CPU_Profile_C/C++)
* [**CPU Profile PLT Relocations**](#_CPU_Profile_PLT)
* [**CPU Profile on Virtual Machine**](#_CPU_Profile_on)
* [**CPU Profile Control APIs**](#_CPU_Profile_Control)
* [**CPU Profile IMIX report generation**](#_CPU_Profile_IMIX)

#### CPU Profile Key Concepts

This section explains various key concepts related to CPU Profiling.

CPU Profiling in CodeXL

The CodeXL CPU Profiler follows a statistical sampling-based approach to gather the profile data periodically. It uses a variety of SW and HW resources available in AMD x86 based processor families. CPU Profiler uses the SW timer, HW Performance Monitor Counters (PMC), and HW IBS feature. The most time-consuming parts of a program have a larger number of samples; this is because they have a higher probability of being executed while samples are being taken by the CPU Profiler.

Sampling Interval

The time between the collection of every two samples is the Sampling Interval. For example, in TBP, if the time interval is 1 millisecond, then roughly 1,000 TBP samples are being collected every second for each processor core.

HW Performance Monitor Counters (PMC)

AMD's x86-based processors have Performance Monitor Counters (PMC) that let them monitor various micro-architectural events in a CPU core. The PMC counters are used in two modes:

* In counting mode, these counters are used to count the specific events that occur in a CPU core.
* In sampling mode, these counters are programmed to count a specific number of events; once the count is reached the appropriate number of times (called sampling interval), an interrupt is triggered. During the interrupt handling, the CPU Profiler collects profile data.

The number of hardware performance event counters available in each processor is implementation-dependent (see the BIOS and Kernel Developer's Guide [BKDG] of the specific processor for the exact number of hardware performance counters). The operating system and/or BIOS can reserve one or more counters for internal use. Thus, the actual number of available hardware counters may be less than the number of hardware counters. The CPU Profiler uses all available counters for profiling.

Time-Based Profile (TBP)

In this profile mode, the profile data is periodically collected based on the specified timer interval. It is used to identify the hot-spots of the profiled applications.

Event-Based Profile (EBP)

In this mode, the CPU Profiler uses the PMCs to monitor the various micro-architectural events supported by the AMD x86-based processor. It helps to identify the CPU and memory related performance issues in profiled applications. CodeXL provides a number of predefined EBP profile configurations. To analyze a particular aspect of the profiled application (or system), a specific set of relevant events are grouped and monitored together. The CPU Profiler provides a list of pre-defined event configurations, such as Assess Performance and Investigate Branching, etc. You can select any of these pre-define configurations to profile and analyze the runtime characteristics of your application. You also can create their custom configurations of events to profile.

This profile mode is supported on the various AMD processor models, such as Family 10h, Family 11h, Family 12h, Family 14h, Family 15h models 00h-0Fh, 10-1Fh, 30-3Fh and Family 16h models 00-0Fh.

In this profile mode, a delay called skid occurs between the time at which the sampling interrupt occurs and the time at which the sampled instruction address is collected. This skid distributes the samples in the neighborhood near the actual instruction that triggered a sampling interrupt. This produces an inaccurate distribution of samples and events are often attributed to the wrong instructions.

Instruction-Based Sampling (IBS)

In this mode, the CPU Profiler uses the IBS HW supported by the AMD x86-based processor to observe the effect of instructions on the processor and on the memory subsystem. In IBS, HW events are linked with the instruction that caused them. Also, HW events are being used by the CPU Profiler to derive various metrics, such as data cache latency.

IBS is supported starting from the AMD processor family 10h.

Event-Counter Multiplexing

If the number of monitored PMC events is less than, or equal to, the number of available performance counters, then each event can be assigned to a counter, and each event can be monitored 100% of the time. In a single-profile measurement, if the number of monitored events is larger than the number of available counters, the CPU Profiler time-shares the available HW PMC counters. (This is called **event counter multiplexing**.) It helps monitor more events and decreases the actual number of samples for each event, thus reducing data accuracy. The CPU Profiler auto-scales the sample counts to compensate for this event counter multiplexing. For example, if an event is monitored 50% of the time, the CPU Profiler scales the number of event samples by factor of 2.

#### CPU Profile Configurations

CodeXL handles all of the details and mechanics of data collection and profile formation.

CodeXL uses statistical sampling to collect and build a program profile. Because all profiles rely on statistical sampling, it is important for the CPU Profiler to take enough samples. The number of samples collected during a session depends on the following:

* how frequently samples are taken (the sampling period or interval)
* the measurement length; that is, the amount of time which takes for the CPU Profiler to take a specific sample
* whether the profiling is system-wide

CodeXL provides the following profile sessions types:

[**Time-Based Profiling**](#_topic_TimeBasedProfiling)

[**Assess Performance**](#_topic_AssessPerformance)

[**Instruction-Based Sampling**](#_topic_InstructionBasedSampling)

[**Investigate Branching**](#_topic_InvestigateBranching)

[**Investigate Data Access**](#_topic_InvestigateDataAccess)

[**Investigate Instruction Access**](#_topic_InvestigateInstructionAccess)

[**Investigate L2 Cache Access**](#_topic_InvestigateL2CacheAccess)

**Cache Line Utilization**

[**Custom Profile**](#_Custom_Profile)

The type and frequency of the profiled events can indicate the presence of a pipeline bottleneck, poor memory access pattern, poorly predicted conditional branches, or some other performance issues. Once hot-spots are found through time-based profiling, EBP and IBS are used identify opportunities for optimization. AMD processors provide a wide range of hardware events that can be monitored and measured. The number of counters and the hardware events that can be measured are processor-dependent.

When compiling your application to use for profiling, consider generating debug information as well. If present, the debug information can help identify profiled areas in the code; however, debug information is not required. Performance data can be collected for an application program that was compiled without debug information; in this case the results displayed by CodeXL are less descriptive. For example, CodeXL displays assembly code rather than function names or source code.

The CPU Profiler measures CPU execution time of a program. Through these measurements it helps you optimize the program. The Profiler records which functions were opened and how long it took to execute each one. CodeXL does this by taking samples. To take a sample, CodeXL interrupts the program’s execution at specified intervals, and logs the state of the program’s call chain.

CodeXL samples all of the various performance monitoring registers to obtain detailed information about the running application or the entire system. While all work can be done through the GUI, profiles also can be collected and analyzed through Visual Studio by using the CodeXL Visual Studio extension.

System profiling identifies a hot-spot anywhere in the system or 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 can be sampled. Any child processes spawned by a profiled process are profiled automatically. During the profiling process, the application to be analyzed is run at full speed on the same machine that is running CodeXL.

Time-based samples (collected at 1 ms intervals on each core) can be used to identify possible bottlenecks, execution penalties, or optimization opportunities. The TBP feature can be used on both AMD and non-AMD processors with an Advanced Programmable Interrupt Controller (APIC) timer. Event-based samples and instruction-based samples can be used to help determine the cause of hot-spots or optimization opportunities. The sampling intervals are weighted so the types of view analysis are valid. The EBP and IBS features are only available on AMD processors.

Call chain sampling collects function call information, including caller-to-called relationships between functions. It is used in conjunction with the selected profile. When call chain sampling is enabled, CodeXL collects information from the run-time call stack of a monitored application process (and child processes) whenever a regular profile sample is taken for the process. When compared to other techniques, such as instrumentation, call chain sampling is a relatively low-overhead approach to the collection of function call information. However, call chain sampling results are subject to statistical variation.

##### Time-Based Profiling

When time-based profiling is enabled and started, CodeXL 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. Post-processing builds up a type of histogram, which describes what the system and its software components were doing. The most time-consuming parts of a program 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.

The frequency of sample taking is controlled by the timer interval. This sometimes is called the "sampling period." The timer interval is 1 millisecond: roughly 1,000 TBP samples are taken each second for each processor core.

The second factor is the length of time during which the samples are taken.. The measurement period depends on the overall execution time of the workload and the way in which CodeXL data collection is configured. Using the CPU Profile Options, CodeXL can be configured to collect samples for all, or part, of the time that the test workload is executing. 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.

Deciding how many samples are enough requires 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 are being aggregated rapidly within the inner loops, and even a fairly short run-time yields a statistically useful number of samples. Other workloads, like transaction processing, usually have just a few inner loops, and the profiles are relatively "flat." For flat workloads, a longer measurement period is required to aggregate samples in code regions of interest.

##### Assess Performance

This pre-defined configuration is intended to get an overall assessment of hardware performance and to give an idea of the possible causes of performance issues.

Hardware Events: (the numeric hardware event codes are the codes the CPU uses to identify these events)

* [0C0] Retired Instructions
* [076] CPU clock cycles not halted
* [0C2] Retired branch instructions
* [0C3] Retired mispredicted branch instructions
* [040] Data cache accesses
* [041] Data cache misses
* [046] L1 DTLB and L2 DTLB misses
* [047] Misaligned accesses

This profile configuration measures eight different events and requires event counter multiplexing. Each event is sampled approximately half of the time (a 50% duty cycle) when four hardware performance counters are available. When using a profile configuration that requires event counter multiplexing, ensure run time is long enough to build up a statistically accurate picture of program behavior.

The available views for profiles with this data are:

* All Data
* Branch assessment
* DTLB assessment
* Data access assessment
* IPC assessment
* Misaligned access assessment
* Overall assessment

##### Instruction-Based Sampling

Instruction-Based Sampling (IBS) identifies and diagnoses performance issues in program hot-spots. It collects data on how instructions behave on the processor and in the memory subsystem; it also provides a range of measurable data for each sample. When running IBS,

* hardware events are linked with the instructions that caused them.
* it produces a wealth of event data in a single test run.
* latency is measured for key performance factors such as data cache miss latency.

IBS provides the most common types of information needed for program performance analysis. It uses a hardware sampling technique to generate event information similar to that produced by event-based profiling. Event-based profiling, however, offers a wider range of events that can be monitored, such as those related to HyperTransport™ links.

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 decode 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.

Note: For more information about instruction-based sampling, see the following documents which are available at [AMD’s Developer Guides & Manuals page](http://developer.amd.com/resources/documentation-articles/developer-guides-manuals/):

* [Software Optimization Guide for AMD Family 16h Processors](http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2012/10/SOG_16h_52128_PUB_Rev1_1.pdf)
* [Preliminary BIOS and Kernel Developer’s Guide (BKDG) for AMD Family 16h Models 00h-0Fh (Kabini) Processors](http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2012/10/48751_BKDG_Fam_16h_Mod_00h-0Fh.pdf)
* [BIOS and Kernel Developer Guide (BKDG) for AMD Family 15h Models 00h-0Fh Processors](http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2012/10/42301_15h_Mod_00h-0Fh_BKDG1.pdf)
* [Software Optimization Guide for AMD Family 15h Processors](http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2012/03/47414_15h_sw_opt_guide.pdf)
* [BIOS and Kernel Developer Guide (BKDG) for AMD Family 14h Models 00h-0Fh Processors](http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2012/10/43170_14h_Mod_00h-0Fh_BKDG.pdf)
* [BIOS and Kernel Developer’s Guide (BKDG) For AMD Family 12h Processors](http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2012/10/41131.pdf)

How IBS Works

IBS provides separate means to sample fetch operations and macro-ops. IBS fetch sampling and IBS op sampling can be enabled and collected separately or together.

IBS Fetch Sampling

This is a statistical sampling method. IBS fetch sampling counts the 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 fetch 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 describe what happened during the fetch operation. Similar to time-based profiling and event-based profiling, CodeXL uses the IBS sample data and information from the executable images, debug information, and source to build a profile IBS for software components executed on the system. IBS is also available in system-wide profiling.

The event data reported in an IBS sample includes the following:

* If the fetch completed or aborted.
* If the 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).
* Fetch latency (number of processor cycles from when the fetch was initiated to when the fetch completed or aborted).

Event-based profiling requires several counters to collect as much information as IBS. 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 occurs when an AMD64 instruction straddles two fetch blocks. In this case, CodeXL associates the IBS fetch sample with the AMD64 instruction in the preceding fetch block.

A fetch can be abandoned before it delivers data to the decoder, or due to a control flow redirection; this can happen 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.

CodeXL identifies killed fetches. The fetch operations remaining after killed fetches are removed from consideration are called **attempted** fetches: these fetches represent valid attempts to obtain instruction bytes.

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.

IbsOps IBS Op Sampling

IBS op sampling operates like fetch sampling. It provides two methods for op selection:

* Cycles mode ‒ IBS hardware counts processor cycles. When reaching the maximum cycle count (the sampling period), IBS tags an available valid op.
* Dispatched op mode ‒ IBS hardware counts ops as they are issued into the pipeline. When the number of dispatched ops reaches the maximum op count (the sampling period), IBS tags the op. Dispatched op mode is preferred because Cycles mode selection is susceptible to delay induced sampling bias.

Note: Some processors do not support dispatched op mode. For more details, see the [BKDG for the AMD processor](http://developer.amd.com/Resources/documentation/guides/Pages/default.aspx) for your platform. 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.

CodeXL uses this and other information to build an IBS profile.

Cycle-based op sampling can be susceptible to timing bias: it can cause ops from some instructions to be selected more often than other instructions. Dispatched op-based sampling is the preferred IBS operating mode because it is not biased by timing.

IBS op samples are taken only for ops that retire. Thus, IBS op event information does not measure speculative execution activity. The cycles-based tagging scheme can introduce statistical bias 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 tags a macro-op and starts counting again from a small, pseudo-random initial count.

IBS op sampling reports the following values for all ops:

* Virtual address of the parent AMD64 instruction from which the tagged op was issued.
* Tag-to-retire time (the number of processor cycles from when the op was tagged to when the op retired).
* 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 mispredictions are attributed to the mispredicted branch, and cache misses are attributed to the AMD64 instruction that caused the cache miss. IBS makes it easier to identify the performance-degrading instructions.

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 or was taken.

IBS also indicates whether a branch operation was a subroutine return, and if the return was mispredicted. Some ops can 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 the following:

* 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).
* Virtual data address for the memory operation.
* Latency when a load misses the DC.

Requests made through the Northbridge produce additional event information:

* Whether the access was local or remote.
* Data source that fulfilled the request.

A full list of IBS op event information appears in the section on IBS-Derive events below. For hardware-level details, see the [BIOS and Kernel Developer's Guide](http://developer.amd.com/Resources/documentation/guides/Pages/default.aspx) (BKDG) for the AMD processor for your platform.

IBS-Derived Events

CodeXL 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 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 that 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.

|  |  |
| --- | --- |
| **Event** | **Description** |
| All IBS fetch samples | 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 |
| 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.) |
| IBS fetch attempted | 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. |
| IBS fetch completed | 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.) |
| IBS fetch aborted | 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. |
| IBS ITLB hit | The number of IBS attempted fetch samples where the fetch operation initially hit in the L1 ITLB (Instruction Translation Lookaside Buffer). |
| IBS L1 ITLB misses (and L2 ITLB hits) | The number of IBS attempted fetch samples where the fetch operation initially missed in the L1 ITLB and hit in the L2 ITLB. |
| IBS L1 L2 ITLB miss | The number of IBS attempted fetch samples where the fetch operation initially missed in both the L1 ITLB and the L2 ITLB. |
| IBS instruction cache misses | The number of IBS attempted fetch samples where the fetch operation initially missed in the IC (instruction cache). |
| IBS instruction cache hit | The number of IBS attempted fetch samples where the fetch operation initially hit in the IC. |
| IBS 4K page translation | 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. |
| IBS 2M page translation | 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. |
| IBS fetch latency | 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. |
| IBS fetch L2 cache miss | The instruction fetch missed in the L2 Cache. |
| IBS ITLB refill latency | The number of cycles when the fetch engine is stalled for an ITLB reload for the sampled fetch. If there is no reload, the latency will be 0. |
| All IBS op samples | 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. |
| IBS tag-to-retire cycles | 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. |
| IBS completion-to-retire cycles | 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. |
| IBS branch op | 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. |
| IBS mispredicted branch op | 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. |
| IBS taken branch op | The number of IBS samples for retired branch operations that were taken branches. |
| IBS mispredicted taken branch op | The number of IBS samples for retired branch operations that were mispredicted taken branches. |
| IBS return op | 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. |
| IBS mispredicted return op | 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. |
| IBS resync op | The number of IBS resync op samples. A resync op is only found in certain microcoded AMD64 instructions and causes a complete pipeline flush. |
| IBS all load store ops | 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. |
| IBS load ops | The number of IBS op samples for ops that perform a load operation. |
| IBS store ops | The number of IBS op samples for ops that perform a store operation. |
| 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). |
| IBS L1 DTLB misses L2 hits | 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. |
| IBS L1 and L2 DTLB misses | The number of IBS op samples where either a load or store operation initially missed in both the L1 DTLB and the L2 DTLB. |
| IBS data cache misses | The number of IBS op samples where either a load or store operation initially missed in the data cache (DC). |
| IBS data cache hits | The number of IBS op samples where either a load or store operation initially hit in the data cache (DC). |
| IBS misaligned data access | 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). |
| IBS bank conflict on load op | The number of IBS op samples where either a load or store operation caused a bank conflict with a load operation. |
| IBS bank conflict on store op | The number of IBS op samples where either a load or store operation caused a bank conflict with a store operation. |
| IBS store-to-load forwarded | The number of IBS op samples where data for a load operation was forwarded from a store operation. |
| IBS store-to-load cancelled | The number of IBS op samples where data forwarding to a load operation from a store was cancelled. |
| IBS UC memory access | The number of IBS op samples where a load or store operation accessed uncacheable (UC) memory. |
| IBS WC memory access | The number of IBS op samples where a load or store operation accessed write combining (WC) memory. |
| IBS locked operation | The number of IBS op samples where a load or store operation was a locked operation. |
| 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). |
| IBS L1 DTLB 4K page | 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. |
| IBS L1 DTLB 2M page | 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. |
| IBS L1 DTLB 1G page | 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. |
| IBS L2 DTLB 4K page | 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. |
| IBS L2 DTLB 2M page | 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. |
| IBS L2 DTLB 1G page | 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 1-GByte page entry for address translation. |
| IBS data cache miss load latency | The total DC miss load latency (in processor cycles) across all IBS op samples that performed a load operation and missed in the data cache. 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 load latency by the number of data cache misses to obtain the average DC miss load latency. |
| IBS load resync | Load Resync. |
| IBS Northbridge 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. |
| IBS Northbridge remote | The number of IBS op samples where a load operation was serviced from a remote processor. |
| IBS Northbridge local L3 | The number of IBS op samples where a load operation was serviced by the local L3 cache. |
| IBS Northbridge local core L1 or L2 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. |
| IBS Northbridge local core L1, L2, L3 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. |
| IBS Northbridge local DRAM | The number of IBS op samples where a load operation was serviced by local system memory (local DRAM via the memory controller). |
| IBS Northbridge 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). |
| IBS Northbridge local APIC MMIO Config PCI | The number of IBS op samples where a load operation was serviced from local MMIO, configuration or PCI space, or from the local APIC. |
| IBS Northbridge remote APIC MMIO Config PCI | The number of IBS op samples where a load operation was serviced from remote MMIO, configuration or PCI space. |
| IBS Northbridge cache modified state | 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. |
| IBS Northbridge cache owned state | 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. |
| IBS Northbridge local cache latency | The total data cache miss latency (in processor cycles) for load operations that were serviced by the local processor. |
| IBS Northbridge remote cache latency | The total data cache miss latency (in processor cycles) for load operations that were serviced by a remote processor. |

##### Investigate Branching

This pre-defined configuration is intended to assist an investigation into branching and near-return performance.

Hardware Events: (the numeric hardware event codes are the codes the CPU uses to identify these events)

* [0C0] Retired Instructions
* [0C2] Retired branch instructions
* [0C3] Retired mispredicted branch instructions
* [0C4] Retired taken branch instructions
* [0C8] Retired near returns
* [0C9] Retired mispredicted near returns
* [0CA] Retired mispredicted indirect branches

The available views for profiles with this data are:

* All Data
* Branch assessment
* Near return report
* Taken branch report

##### Investigate Data Access

This pre-defined configuration helps investigate data locality and poor DTLB behavior.

Hardware Events: (the numeric hardware event codes are the codes the CPU uses to identify these events)

* [0C0] Retired Instructions
* [040] Data cache accesses
* [041] Data cache misses
* [042] Data cache refills from L2 or Northbridge
* [045] L1 DTLB miss and L2 DTLB hit
* [046] L1 DTLB and L2 DTLB misses
* [047] Misaligned accesses

The available views for profiles with this data are:

* All Data
* DTLB assessment
* DTLB report
* Data access assessment
* Data access report
* Misaligned access assessment

##### Investigate Instruction Access

This pre-defined configuration helps investigate instruction fetches with poor L1 locality and poor ITLB behavior.

Hardware Events: (the numeric hardware event codes are the codes the CPU uses to identify these events)

* [0C0] Retired Instructions
* [080] Instruction Cache fetches
* [081] Instruction Cache misses
* [084] L1 ITLB miss and L2 ITLB hits
* [085] L1 ITLB miss and L2 ITLB miss

The available views for profiles with this data are:

* All Data
* ITLB report
* Instruction Cache Report

##### Investigate L2 Cache Access

This pre-defined configuration helps investigate memory access operations with poor L2 cache locality.

Hardware Events: (the numeric hardware event codes are the codes the CPU uses to identify these events)

* [0C0] Retired Instructions
* [07D] Requests to L2 cache
* [07E] L2 cache misses
* [07F] L2 fill/writeback

The available views for profiles with this data are:

* All Data
* L2 Access Report

##### Cache Line Utilization

This feature is a first step towards providing data-centric application profiling capabilities. This feature models the behavior of the processor L1 data cache and uses the Load and Store Instruction-Based Sampling (IBS) records to provide a measure of how efficiently an application utilizes the L1 data cache.

A cache is a relatively small amount of on-chip memory which is extremely fast compare to main memory. When the processor needs to access a location in main memory, it first checks whether a copy of that data is in the cache. If the data is present in cache, it is called a cache hit and the processor immediately accesses the data from cache. If the data is not in the cache, it is called a cache miss. In this case processor has to wait for the data to be fetched from main memory before it can continue to execute. All of the data required by all of the processes running on a processor cannot simultaneously fit in the cache, so the processor removes, or evicts, data from the cache when new data is needed and the cache is full. Data is transferred between memory and cache in blocks of fixed size, called cache lines.

The cache misses directly influence the performance of the application. Having the data in the cache when the processor needs it is one way to optimize performance of an application. Additionally, because cache size is small, it is desirable to fill the cache with data that will be used before it is evicted from the cache.

AMD processors have a separate instruction and data cache per core (L1 (Level 1) instruction and L1 data caches) as well as a unified L2 (per module) cache and L3 (per-chip) cache. However, CLU models the L1 data cache only. CLU measures how much of a cache line is used (read or written) before it is evicted from the cache. The percent cache line utilization is defined as percentage of number of bytes in the cache line had been accessed before the cache line has been evicted.

A low CLU value implies that the cache is being filled with data that is never or less accessed before it is evicted, implying cache capacity pressure, as well as main memory bandwidth pressure (reading data from main memory that is not accessed before being evicted).

A high usage percentage (CLU) means that the application is properly exploiting spatial and temporal locality of its data. Ideally, one would like to have 100 percent CLU. Practically speaking however, a good CLU is about 20 to 30 percent, primarily due to the sampling nature of the core in its collection of the load and store data.

Note: See the [BIOS and Kernel Developer's Guide for AMD Family 10h Processors (order #31116) for detailed information about caches.](http://developer.amd.com/Resources/documentation/guides/Pages/default.aspx)

The following table describes the data that can be shown for each module, function, source and disassembly

|  |  |
| --- | --- |
| **Event** | **Description** |
| Cache Line Utilization Percentage | The cache line utilization percentage for all cache lines on all cores accessed by this instruction / function / module. |
| Line Boundary Crossings | The number of accesses to the cache line that spanned two cache lines. This happens when an unaligned access is made that causes two cache lines to be touched. |
| Bytes/L1 Eviction | The number of bytes accessed between cache line evictions. |
| Accesses/L1 Eviction | The number of accesses (loads plus stores) to a cache line between evictions. |
| L1 Evictions | The number of times a cache line was evicted where this instruction depended on the data in the cache line. |
| Accesses | The total number of loads and stores samples for this instruction / function / module. |
| Bytes Accessed | The total number of bytes accessed by this instruction / function / module. |

##### Custom Profile

This configuration is intended for advanced users who know what hardware events or combinations of events are important for specific analysis.

It allows a combination of hardware events, timer samples, and instruction-based sampling events. At least one event must be monitored for CodeXL to make the profiling.

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 such as instruction and data caches, translation lookaside buffers, and branch history tables. Extremely small sampling periods also can cause system instability. Start off conservatively and slowly decrease the sampling period for an event until the appropriate volume of samples is generated.

* **Warning** You can demand too much sampling, causing the system to hang or crash. If this occurs, the easiest solution is to increase the sample intervals for the most popular events.

A factor when choosing the sampling period for an event is the workload behavior. Some workloads are CPU-intensive; other workloads are memory-intensive. Some workloads can be CPU-intensive and require high memory bandwidth to stream data into the CPU. For example, a CPU-intensive application that does not accesses memory very often causes relatively few data-cache miss events. The characteristics of the workload can 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, requiring changes to the sample period.

The available views for the profile data depend on the events selected. The events are selected through the Error! Reference source not found.Dialog.

### CPU Profile Session

A CodeXL CPU profile session collects profile data of a single execution of the profiled application. The profile session results can be viewed and analyzed using CodeXL tabbed views. After making the recommended changes to the profiled application, you can execute the same application and analyze the new bottlenecks after the change.

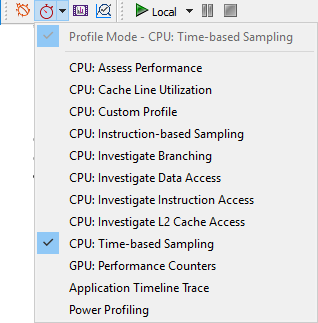
To create and run a CPU profile session in CodeXL requires [**Executing CPU Profile Session**](#_topic_ExecuteCPUProfileSession)Other options can be configured in the project and global settings:

1. [**Changing CPU Profiling Global Settings**](#_CPU_Profile_Global)
2. [**Changing CPU Profiling Project Settings**](#_topic_CPUProfileProjectOptions)
3. [**Changing CPU Profiling Configuration Events**](#_topic_AdvancedCPUProfilingConfiguratio)

#### Execute a CPU Profile Session

To run a CPU Profile session:

1. Open or create a CodeXL project



1. Select a CPU profile type. See detailed information for each of CodeXL supported [**CPU Profile Types**](#_topic_CPUProfileConfigurations)   
   In the Active Mode toolbar, click the Profile Mode toolbar button to change the mode to Profiling (Profile > Profile Mode). Select any CodeXL CPU type of profile (Profile > CPU: Assess Performance for example).
2. Use the [**CPU Profile Setting dialog**](#_topic_CPUProfileProjectOptions) to configure the profile session parameters.
3. Click the Start CodeXL Profiling toolbar button, , to start profiling.
4. Optional: Pause / Stop the data collection of the profiled application using the execution toolbar buttons: .   
   Stopping/pausing the profile session is optional. You can let the application run to its end and then the profiling session will automatically end. For long running applications, pausing allows you to control when the profiling data collection occurs so it matches the stage in your application execution that you want to profile.
5. When the profiled application execution is over, CodeXL processes the data collected, and a profile session window is opened. See the [**View and Analyze the Profile Session Data Analysis**](#_topic_CPUProfileDataAnalysis)section.

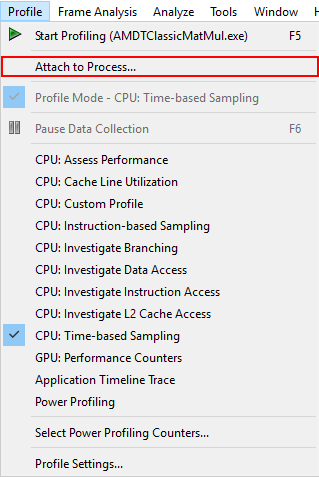
#### Attach to Process

To start a CPU Profile session of an already running process:

1. Select a CPU profile type. See detailed information for each of CodeXL supported [**CPU Profile Types.**](#_topic_CPUProfileConfigurations)

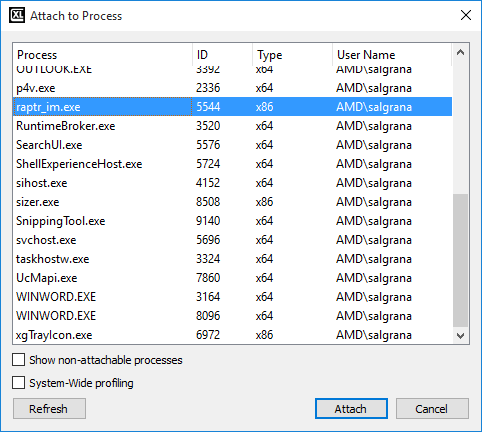
In the Active Mode toolbar, click the Profile Mode toolbar button to change the mode to Profiling (Profile > Profile Mode). Select any CodeXL CPU type of profile (Profile > CPU: Assess Performance, for example).

1. Use the [**CPU Profile Setting dialog**](#_topic_CPUProfileProjectOptions) to configure the profile session parameters. Note that if a CodeXL project is not opened when changing the profile settings, then a generic new project is automatically created.
2. Open the “Attach to Process…” dialog from the menu bar (Profile > Attach to Process…). Note that if a CodeXL project is not opened (or created) when the profiling session starts, then a generic new project is automatically created.



1. Select the desired process from the list of attachable processes and click the “Attach” button to start profiling.
2. Optional: Pause / Stop the data collection of the profiled application using the execution toolbar buttons: .   
   Stopping/pausing the profile session is optional. You can let the application run to its end and then the profiling session will automatically end. For long running applications, pausing allows you to control when the profiling data collection occurs so that you can match the stage in your application execution that you want to profile.
3. When the profiled application execution is over, CodeXL processes the data collected, and a profile session window is opened. See the [**View and Analyze the Profile Session Data Analysis**](#_topic_CPUProfileDataAnalysis)section.

Attach to Process dialog



The dialog shows all the processes in the system, which are divided into 2 groups:

1. **Attachable processes:** These are the processes which the current user may attach to. Note that a user may not attach to other users’ processes without having administrative privileges.
2. **Non-attachable processes:** These processes cannot be attached to due to certain privilege protections. These processes are not viewed by default, and are viewed only if the “Show non-attachable processes” checkbox is checked.

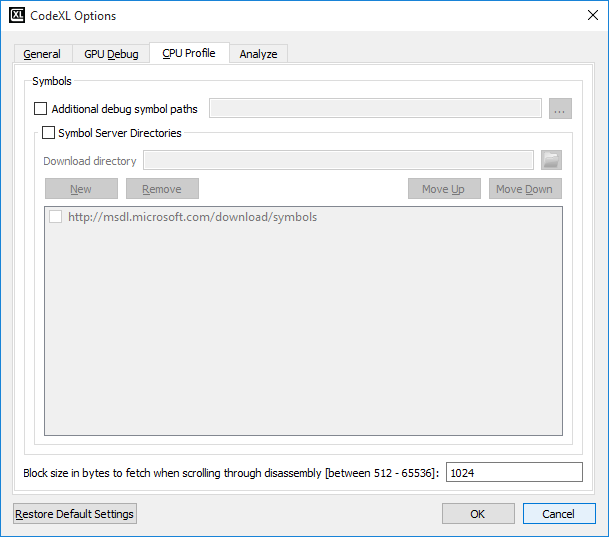
The list of processes is static, and may be refreshed by clicking the “Refresh” button (or reopening the dialog).

#### CPU Profile Global Settings

The CPU Profile global settings are set when the project is initially created. These settings affect every session and control how AMD CodeXL displays profile data.

To set the options:

1. In the CodeXL menu, click **CodeXL Options**.   
   The Edit CodeXL Global Settings dialog box is displayed.
2. Select the CPU Profile tab.   
   The CPU Profile tab controls the display of source files and symbolic information. See the description for each of the global profile settings below.
3. Do one of the following:  
   Click **OK** to activate the new options and close the dialog box,   
   Click **Restore Default Settings** to reset the dialog box selections to the system default, or   
   Click **Cancel** to close without committing any changes.



|  |  |
| --- | --- |
| Additional Debug Symbol Paths | Select least one symbol server. |
| Symbol Server Directories | Select directories for the symbol server. |
| Block size in bytes to fetch when scrolling through disassembly | The size of the data block that is being automatically fetched when scrolling through disassembly. |

#### CPU Profile Project Options

Use the CodeXL Project Settings dialog to configure the current project CPU profile settings:

General Settings Page

To configure a specific application for profiling, enter the path to the application executable in the ‘Executable Path’ field.

This field can be left empty if you intend to perform System Wide profiling.

To profile a Java application:

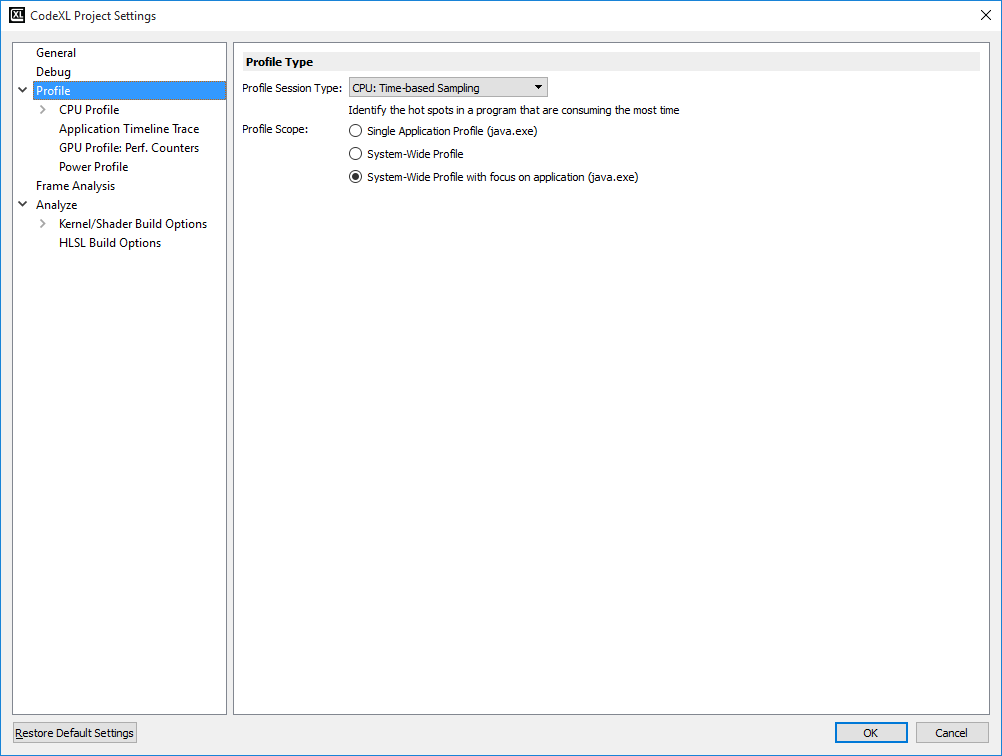
* Enter the path of the Java runtime executable in the ‘Executable Path’ field, e.g. “C:\Program Files\Java\jre7\bin\java.exe”
* Enter the path of the java classes top folder in the “Working Directory” field, e.g. “C:\Scimark2”
* Enter the name of the main java class in the command line arguments, e.g. “jnt.scimark2.commandline”

The screenshot below shows an example of configuring a Java application for profiling.

Profile Settings Page

Click File -> Project Settings and select the Profile tree node.

The “Profile Type” settings page will help you set the profile type of the current project and the profile scope:

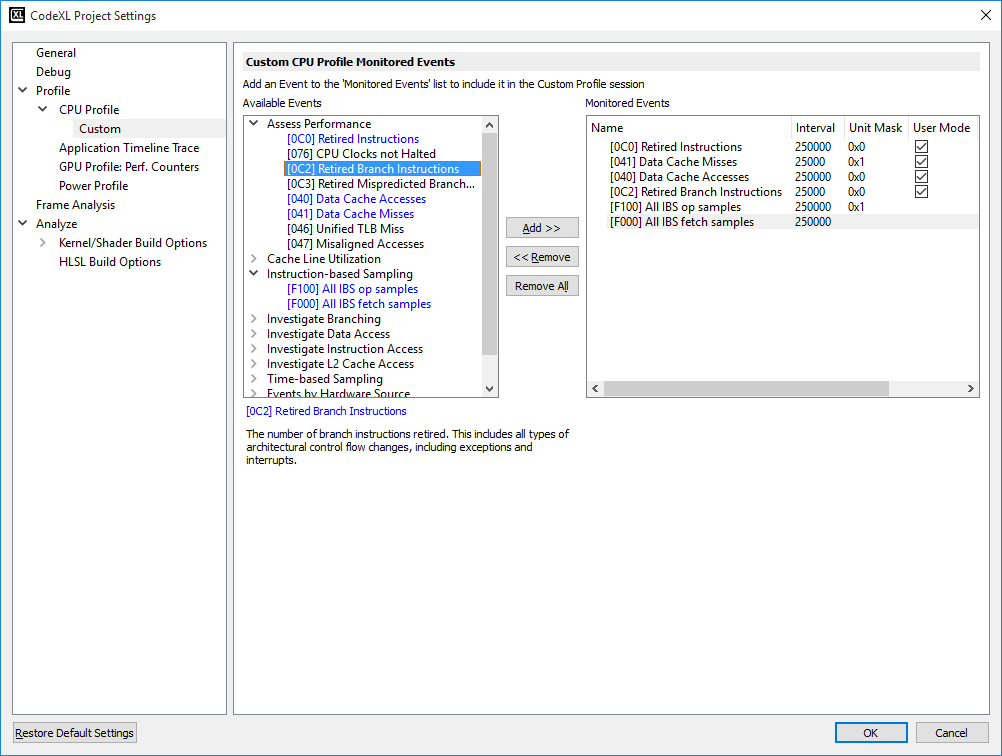


|  |  |
| --- | --- |
| Profile Session Type | Displays the currently selected [**Profile Type.**](#_topic_CPUProfileConfigurations)The combo box contains both CPU and GPU profile types. Use this to select which type of data you want to collect while running the next profile session.  Use the text below the profile type combo box to get a description of each of the profile types. |
| Profile Scope | Defines the scope that the next profile session will monitor.  Single Application – the profile session will collect data for only the profiled application (defined in the “General” settings page).  System-Wide Profile – the profile session will collect data for each of the running processes in the system.  System-Wide Profile with focus on application – the profile session will collect data for each of the running processes in the system. The profile session will also collect call stack details for the focused application (defined in the “General” settings page). |

CPU Profile Settings Page

Click Profile -> Profile Settings to open this settings page.

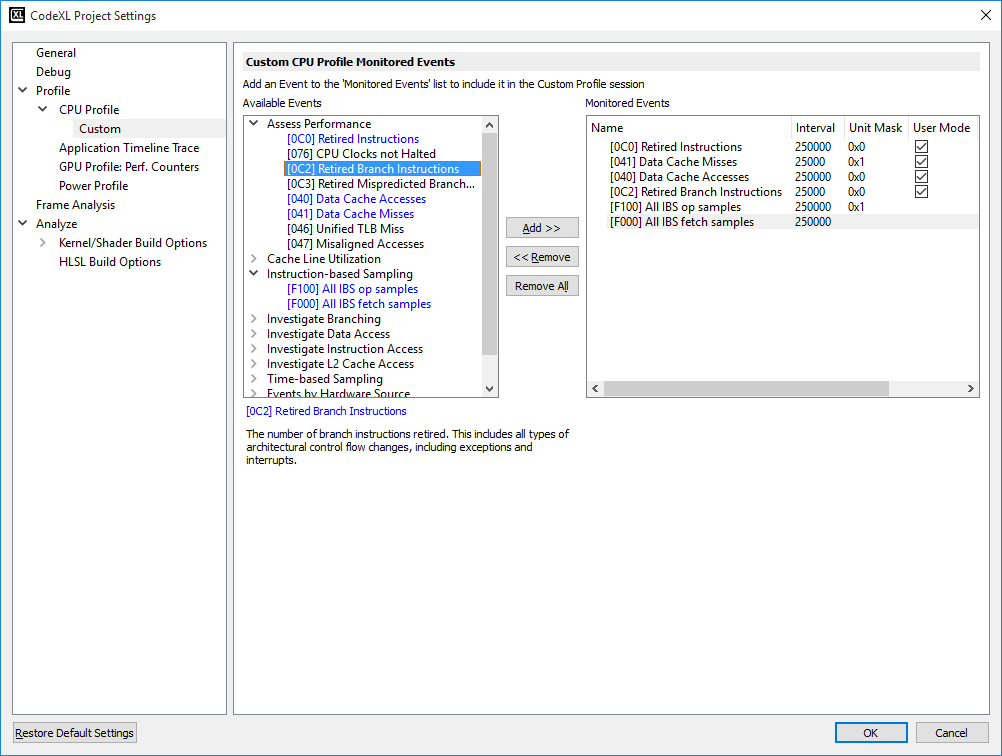
The CPU Profile settings page contains CPU profile specific configurations.



|  |  |
| --- | --- |
| Collect call stack details | Checking this option will set CodeXL to collect call stack details while profiling.  **Note**: Call stack collection has higher overhead compared to sessions without call stack sampling, because stack unwinding is done whenever a sample is taken for the target process. The unwind operation, combined with the larger amount of data that must be written to the trace file, creates the higher overhead. Also, because of OS limits, the complete call stack might not be available. Set the appropriate “Call stack collection depth” and “Call stack every” options, to balance between performance overhead and statistical accuracy. |
| Collect for code executed in | Limits the collection of call stacks only for code running in User space, Kernel space or both (User space and Kernel space). |
| Collect call stack every | This parameter will define the frequency of call stack collection. |
| Call Stack Collection Depth | Defines the level of depth for the collected call stack. A higher depth will require performance overhead, but the data of the collected call stack will be more accurate.  Options are: Minimal / Low / Medium / High / Maximal.  Selecting one of these options will sample the call stack with an up to the depth of 2 / 32 / 64 / 128 / 392 levels.  **Note:** This option is set separately for Time-Based Sampling and other CPU Profiling session types. |
| Reproduce missing call stack info | Perform additional analysis to overcome frame-pointer omission (FPO) in 32-bit apps and lack of unwind info in 64-bit. The profiler will store additional data during the profile session and require more time during post-session processing.  **Note:** This option is set separately for Time-Based Sampling and other CPU Profiling session types. |
| Collection Schedule | Configures the schedule of collecting data during the profile session execution.  Throughout entire duration – collect data throughout entire profile session execution  Start profile with collection paused – the profile session will start with no data collection. Use the “Pause” button to resume data collection on the profiled application.  Scheduled – set the specific timing for data collection during the execution of the profile session. |
| Start data collection after | Start data collection with a delay of ‘X’ seconds from the start of the profile session |
| End data collection after | When selected, the profile session data collection will end after ‘X’ seconds. |
| Then, terminate the process | When checked, after ‘X’ seconds selected to end the data collection after, the profiled process will be terminated by CodeXL. |
| Profile hardware scope | Use the tree structure of the existing hardware cores, or the affinity mask, to define the cores on which the profiled data will be collected. |
| Restore Default Settings | Restore the original default settings. |

CPU Profile Custom Settings Page

Click Profile -> Profile Settings to open this settings page. Select “Profile -> CPU Profile -> Custom” tree node to go to the custom settings page.



This settings page configures the list of events collected while running a session of type “Custom Profile”.

**Available Events** lists all the available events.   
**Monitored Events** lists the events which are selected for monitoring.   
Use **Add**, **Remove**, **Remove All** buttons to add the selected event(s), remove the selected event(s), or remove all the selected events.

1. Some events have a checkbox:

* **Usr** Enables the collection of user-level samples for an event.
* **Os** Enables the collection of operating system-level samples for an event.
* **Edge** Enables the edge- and level- detection that control the way an event signal is sensed; this affects the way an event is counted in a performance counter.

The available events depend on the CPU hardware in your system. Note that only one **Timer** event, **IBS all op** samples event, or **IBS fetch** sample event can be monitored at a time.

|  |  |
| --- | --- |
| Available Events | Lists the available hardware events for profiling, nested within the configurations. |
| Add / Remove buttons | Adds to, or removes from the Monitored Events table the selected available event or configuration. |
| Monitored Events | A list containing the selected events for data collection in the “Custom Profile” sessions.  A monitored event item in this list will contain:  **Name -** The name of the monitored event.  **Interval** – the period of how often a sample is to be taken (for counting occurring events).  **Unit Masks -** Used to specify the unit mask setting for the selected event. Each bit set has a different meaning. The **Event Settings** pane gives you a description of the current setting.  **Usr -** Enables collection of user-level samples for an event.  **Os -** Enables collection of operating system-level samples for an event.  **Edge** - Edge detection and level detection control the way an event signal is sensed; it affects the way an event is accumulated as a count in a performance counter. The occurrence of an event (a hardware condition) is asserted as a physical hardware signal. An event has a duration that can be as short as a single CPU clock cycle or it can be several cycles long.  When **OK** is clicked and the Edge checkbox is checked, the "Custom Profile" configuration is selected automatically, and profiling can start immediately. |

Notable Available Events

|  |  |
| --- | --- |
| [E000] Timer event | Hardware APIC timer event. The default is 1ms. The minimum interval is 0.1ms. |
| [F000] IBS fetch samples | Determines how often an IBS fetch sample is taken. IBS fetch sampling counts completed fetches to determine when the next IBS fetch sample is taken. |
| [F100] IBS all op samples | Determines how often an IBS op sample is taken. When the Unit Mask is 0x0 (Count clock cycles), IBS op sampling counts processor cycles to determine when the next IBS op sample is taken. When the Unit Mask is 1 (Count ops dispatched), IBS op sampling counts dispatched ops to determine when the next IBS op sample is taken. Dispatched op counting is the preferred mode because profiles produced through cycle counting can be biased by instruction timing. |
| Events by Hardware Source | Performance monitoring counter events that vary according to the system's hardware. The individual descriptions are displayed when the event is selected. |
| HardwareEventNotes Hardware Performance Counter Notes | A hardware event can be added multiple times, but the unit mask, Usr, or Os settings must be different. When an available event or configuration is selected, a description is shown below the list. Most event details can be directly edited within the table by clicking on the detail to change. When an event is selected, the unit mask details are also shown below the Monitored Events table.  The presence of a hardware condition is asserted when the event signal is high. Absence of the condition is asserted when the event signal is low. When edge detection is used, each low-to-high transition of the event signal is counted as a single event; that is, the performance counter is incremented by one. When level detection is used, the level is sensed during each clock cycle, and the performance counter is incremented by one for each cycle during which the event signal is asserted.  The typical setting for an event is level-detect (the Edge box is unchecked). For example, if the event signal represents the "NOT HALTED" CPU state, then the performance counter counts the number of CPU clocks that the CPU spent in the "NOT HALTED" state. This configuration corresponds to the conventional "CPU Clocks Not Halted" event.  Performance counters count either specific processor events or the duration of events. The "Dispatch Stalls" counter event, for example, measures the number of processor cycles when the instruction decoder has stalled for any reason. Edge detection and level detection can be used to determine the average number of cycles per stall by:   * Counting the number of dispatch stall cycles (level detection). * Counting the number of dispatch stalls (edge detection). * Dividing the number of stall cycles by the number of stalls. |

### CPU Profile Data Analysis

The data and source results of a profile session are displayed in the form of tabulated information and annotated source code.

Use the CodeXL Explorer to navigate between tabs of the same sessions, and between different sessions.

Profile session results contain several pages that can be accessed through the CodeXL Explorer and by using the context menus for any of the profile session data tables.

The following views can be displayed for each session:

[**CPU Profile Session Explorer**](#_topic_CPUProfileSessionExplorer)

[**Overview Page**](#_topic_ProfileSessionOverviewPage)

[**Profile Session Modules View**](#_topic_ProfileSessionModulesView)

[**Profile Session Functions View**](#_topic_ProfileSessionFunctionsView)

[**Profile Session Call Graph View**](#_topic_ProfileSessionCallGraphView)

[**Profile Session Source or Disassembly View**](#_topic_ProfileSessionSourceDASMView)

[**Profile Session Display Settings**](#_topic_ProfileSessionDisplaySettings)

[**Import a Profile Session**](#_topic_ImportingProfileData)

[**Save CPU Profile Data**](#_topic_SavingProfileData)

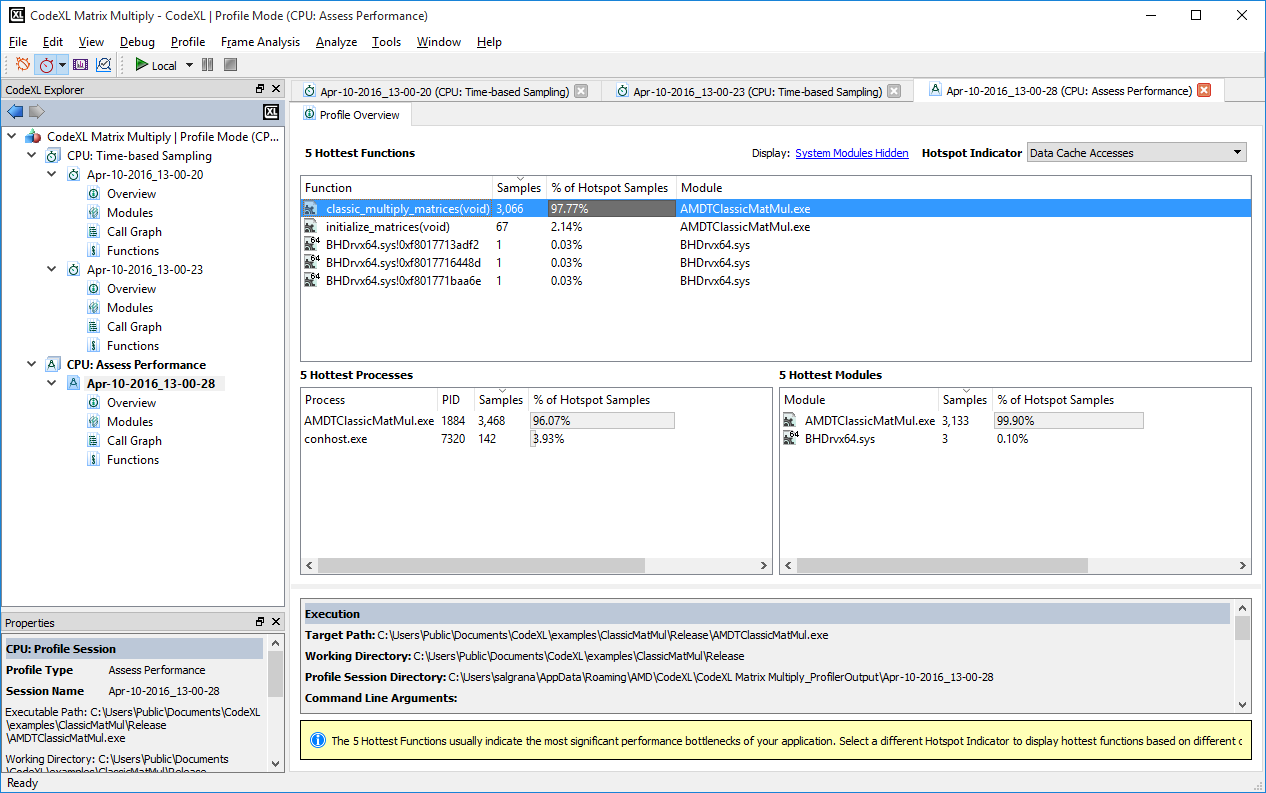
#### CPU Profile Session Explorer

The CPU Profiling project consists of the CodeXL Explorer pane, the data pane, menu and toolbar commands, and subsequent tabs and views that are displayed during the profile sessions.

The CodeXL Explorer pane lists the profile sessions collected for the current project, organized by profile session type.

The CodeXL Explorer lets you:

* Double-click on a session node in the tree, to open it in the data pane.
* Right-click on a tree node to delete, rename, or import sessions.
* Right-click on a session tree node to browse the folder that contains the session. In the screenshot below you can see the CodeXL explorer for “classic” project. The session in bold is the currently selected profile session, and the profile type in bold font (CPU: Assess Performance) is the profile type for the currently selected session.



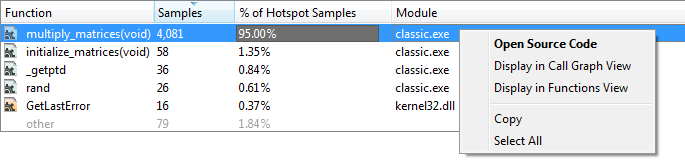
Session Views Right-Click Menus

Another way to navigate between the profile session views is to use the session views' right-click menus. For example, in the screenshot below, the function "multply\_matrices" is marked as a hot-spot in the overview page. Right-click on the function to see the following context menu commands:

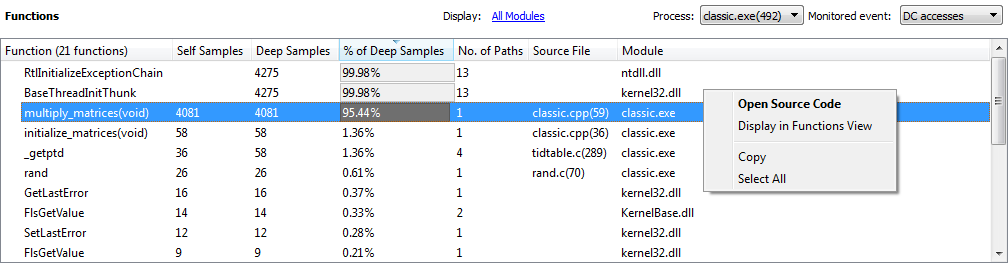
**Opening Source Code** opens the source code for "multply\_matrices" to show how the data samples are distributed within the selected function lines.

**Display in Call Graph View** opens the call graph view to show the "multply\_matrices"' call paths.

**Display in Functions View** opens the functions view to see how the data is distributed within the functions in this module (classic.exe).

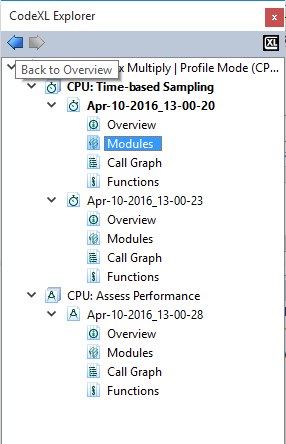


After clicking the "Display in Call Graph" command, the call graph view is opened, with the function "multiply\_matrices" selected. When right-clicking on the function, a context menu is opened with the option to either display the function in source code or in functions view.



CodeXL Explorer Navigation Arrows

The CodeXL Explorer arrows record the navigation to each of the session views that were opened. Use these arrows to go back and forward between the displayed session views.



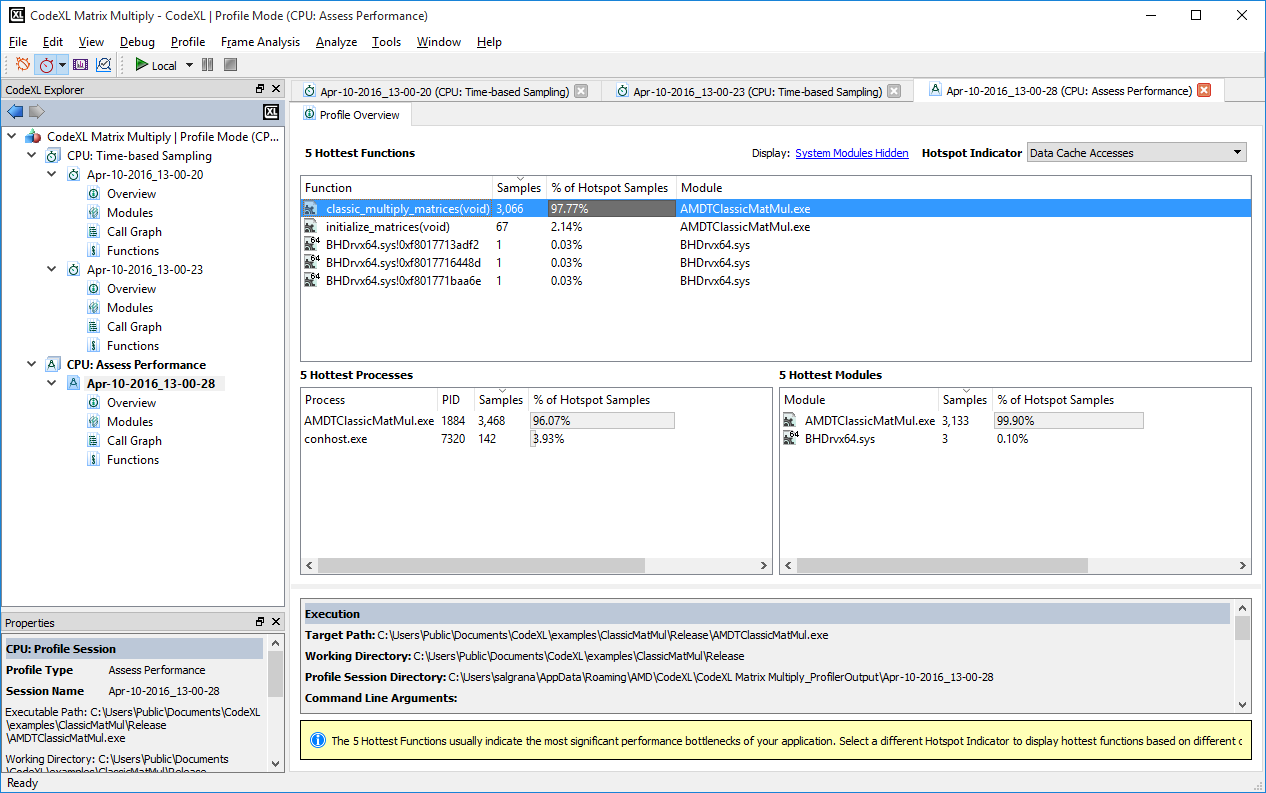
#### Profile Session Overview Page

The CPU Profile Session Overview Page describes an overview of the data collected during the profile session.

The top section of the overview page contains the following data tables: Hot Spot Functions table, Hot Spot Modules table and Hot Spot Processes table. Note that the Processes table is only displayed when running a system-wide profile session.

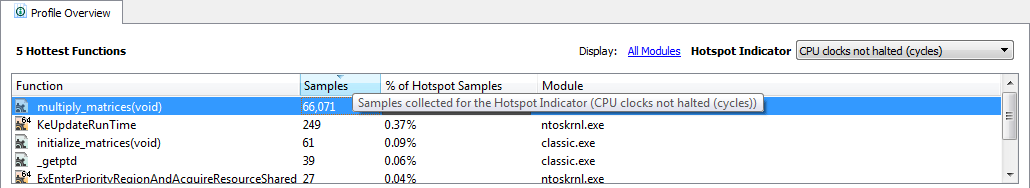
The overview data tables can get a zoom-out image of the performance of the executed application. Use these tables to review the summary of hot spot locations in your application. The tables display the data organized by a 'hot spot indicator'.

The bottom section of the overview page contains general information for this session.



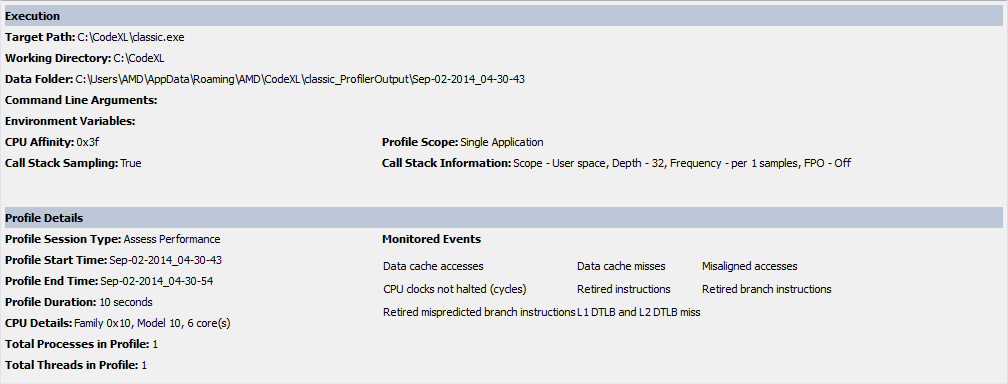
Hot Spot Indicator

The screenshot above displays an Assess Performance session overview. The combo box at the right side of the view is a hot-spot indicator combo box. Selecting another field in this combo box causes the data in the view to be re-calculated. The samples column contains the hot spot functions/modules/processes calculated with the new indicator. The screenshot below displays the same session overview, after selecting "CPU clocks not halted (cycles)" in the hot spot indicator combo box. The tables are updated to display hottest functions / modules/ processes according to each of the monitored event / data field listed in the combo box. The table is updated with the CPU clocks sample count and percent, and the "Samples" column tooltip now specifies the CPU Clocks event.



Session Properties Section

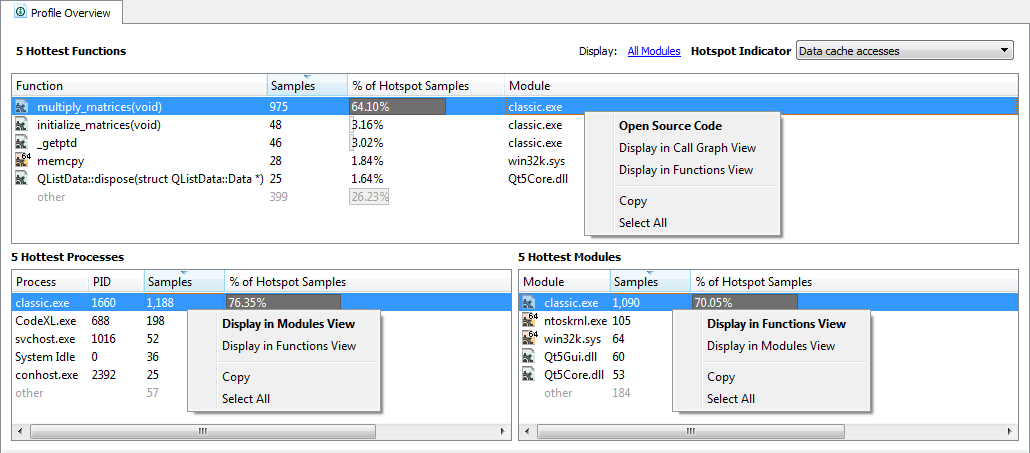
The bottom section of the overview page contains the profiled session properties.



|  |  |
| --- | --- |
| Execution | Contains the session execution properties:   * Target Path, Working Folder, Data Folder, Command Line Arguments and Environment Variables. These session settings are general CodeXL project settings. Use the [**CodeXL Project Settings Dialog**](#_topic_ProjectSettings) to configure them. * CPU Affinity, Call Stack Sampling, System-Wide Profile, Call Stack Information -these settings are CPU profile project settings. Use the [**CodeXL CPU Profile Project Settings Dialog**](#_topic_CPUProfileProjectOptions) to configure them. |
| Profile Details | Contains the following information of the profile session: Profile session type (See [**CPU Profile Types**](#_topic_CPUProfileConfigurations) for details), Profile start-end time, Duration, CPU Family and Total Amount of Processes and Threads. This section also contains a list of monitored events collected for this session. |

Navigating to other views

The Overview page describes a zoomed-out image of the profiled session. For more information about bottle necks in your application, use [**Modules View,**](#_topic_ProfileSessionModulesView) [**Functions View**](#_topic_ProfileSessionFunctionsView) , and [**Call Graph View.**](#_topic_ProfileSessionCallGraphView) You can also open a [**Source-Code View**](#_topic_ProfileSessionSourceDASMView) to see a source-line level performance data. These views can be opened using the [**CodeXL Explorer Tree**](#_topic_CodeXLExplorer) . Each of the tables in the overview page implements a right-click menu that lets you navigate between session views.



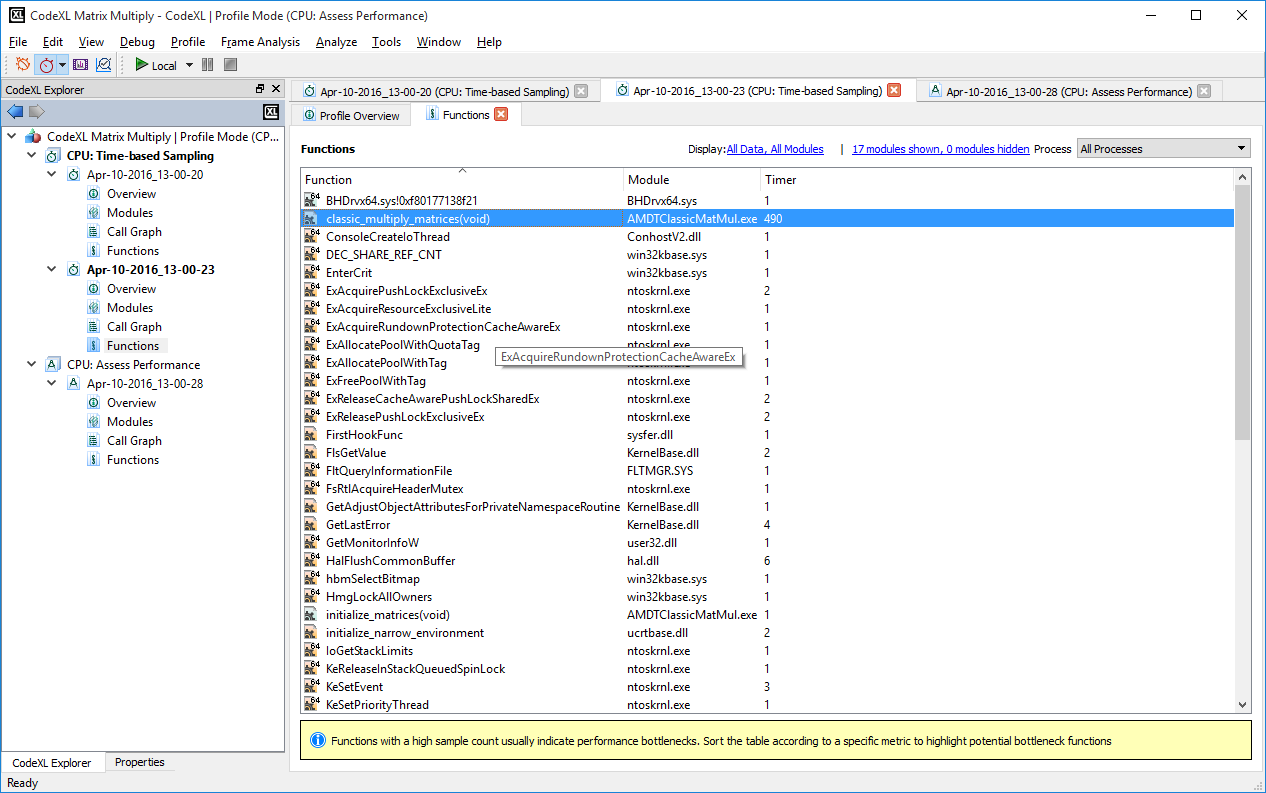
Per-Process vs System-Wide Profile Session

A CPU Profile session can be:

* Per-process ‒ Only the target application is profiled. This is the default.
* System-Wide ‒ Samples are taken for all running processes in the system. Use the [**CodeXL CPU Profile Project Settings Dialog**](#_topic_CPUProfileProjectOptions) to check / un-check the system-wide profile option.

#### Profile Session Modules View

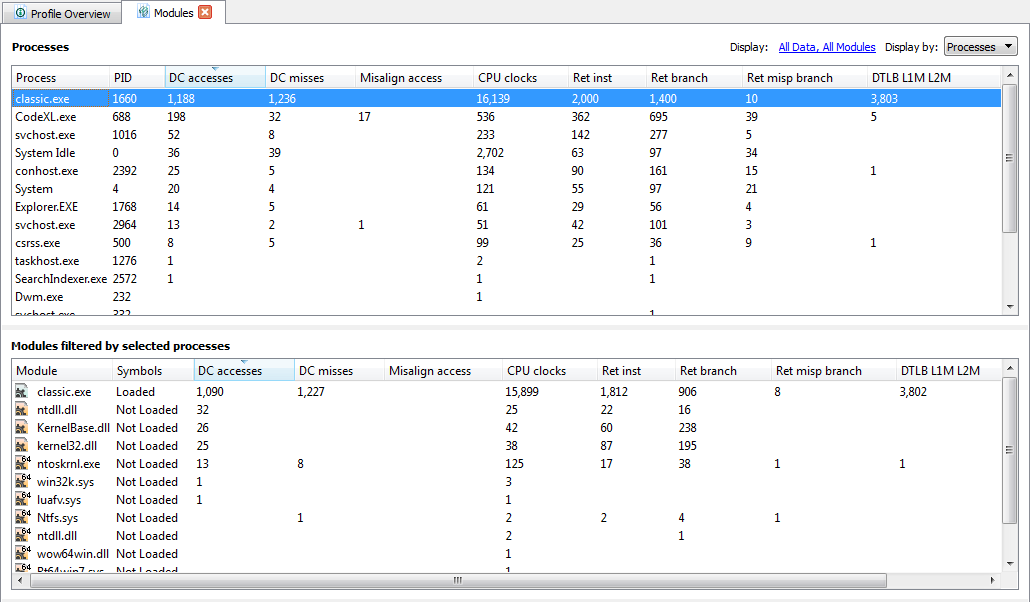
This displays a module-by-module detailing of performance data. Use this view to see the distribution of hot spots among the modules of your application. The data displayed in this view is collected for the session profile type. The screenshot below displays an Assess Performance profile session, with a single process (classic.exe). Modules with a high sample count usually indicate performance bottlenecks. Sort the modules table according to a specific counter to highlight potential bottlenecks.



The modules view displays two tables: processes and modules. The following screenshot shows the “classic” project in a system-wide session. The combo-box on the right side of the view lets you select display processes or modules in the top table.

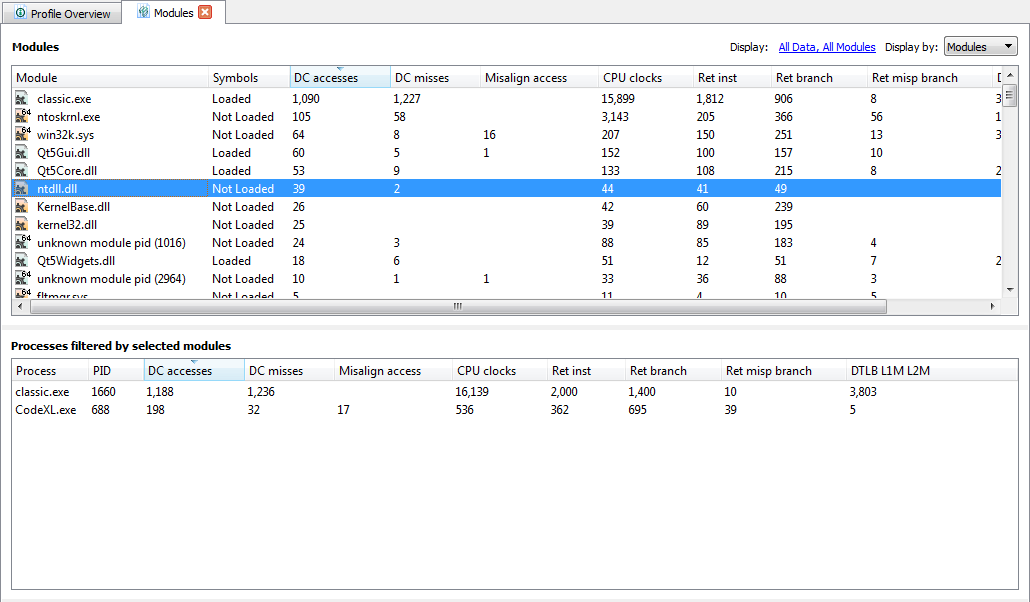
Display Modules View by Processes

The top table displays processes. Select one or more processes in the top table, and the bottom table displays only the modules used by the selected processes.



Display by Modules

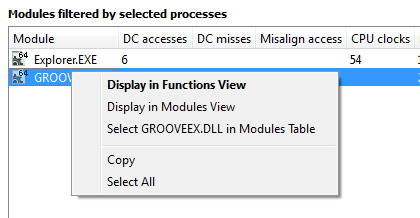
The top table displays modules. Select one or more modules in the top table, and the bottom table displays only the processes used by the selected modules.

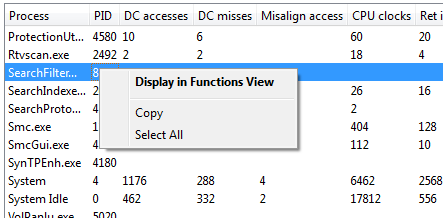


Modules View Context Menus

The following two screenshots display the context menus for the modules and processes tables. The context menus lets you:

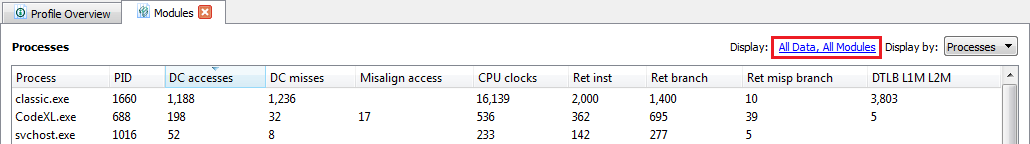
* Display in Functions View ‒ Displays the selected module / process in the functions view.
* Select "ModuleName.dll" in modules table ‒ Switches to "Display by Modules" and selects ModuleName.dll in the modules table.





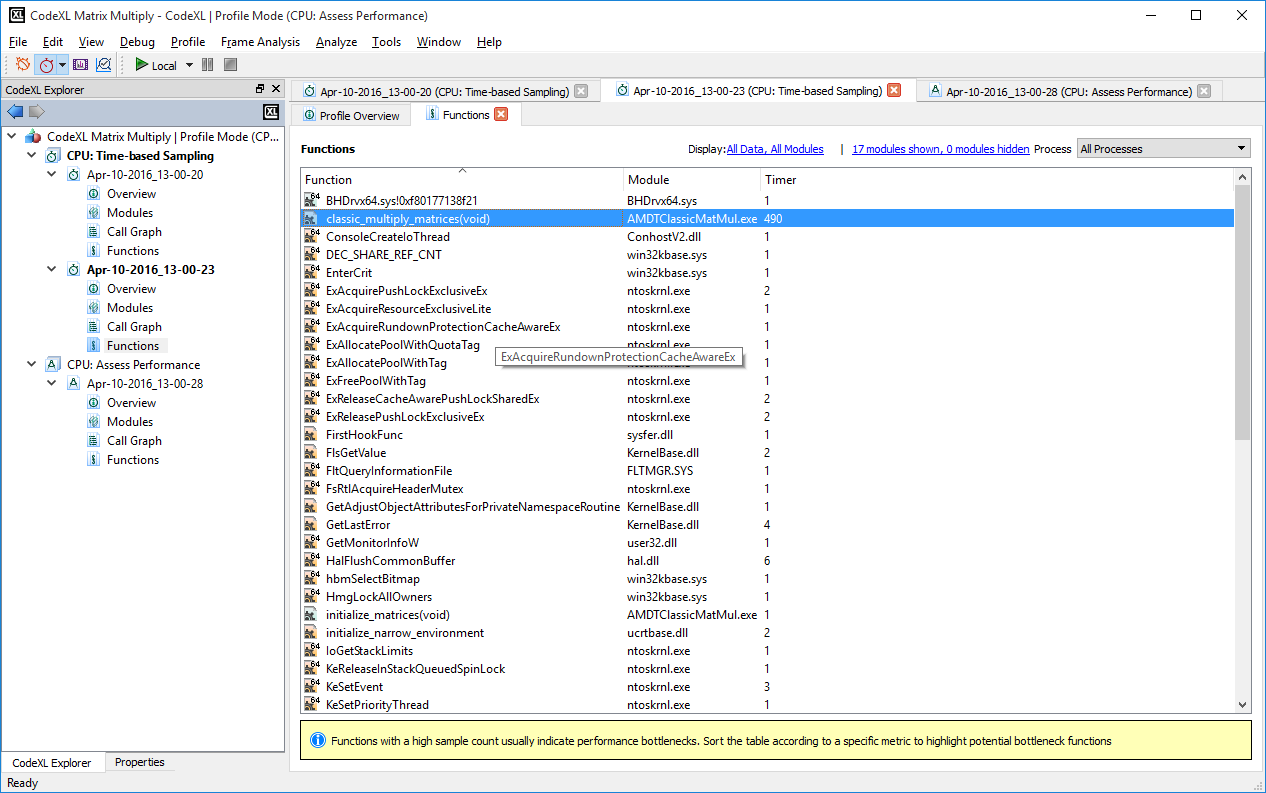
Modules View Display Settings

Use the display filter link to open the [**Display Settings Dialog**](#_topic_ProfileSessionDisplaySettings) and change the properties of the currently displayed session.



#### Profile Session Functions View

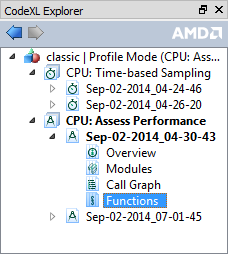
Function view displays list of functions called during profiling of the current session in a table.



Open Functions View

This view can be opened in any of the following ways.

* Double-click the **Functions** node of a session. It opens the functions view with Process selected as **All Processes**. If the view was already there, it just opens the view tab.



* Double-click the process table in the Profile Session Overview Page. This opens the Function view selecting that process. All the functions from that process are listed. This way of opening the Function view is possible only when multiple processes exist.
* Double-click the Module table in Profile Session Overview Page. This opens the Function view selecting only that module.
* Double-click the process table in Profile Session Modules View. This opens the Function view selecting that process.
* Double-click the Module table in Profile Session Modules View. This opens the Function view selecting only that module.
* Selecting the "Display in Functions View" option from context menu of any of the four tables of the Call Graph View. This opens the Function view selecting the function that was selected in Call Graph view on the current table.

Columns

This table shows two columns:

* Function ‒ Contains the name of the function, if available, or a **NO\_SYMBOL** string if the function name could not be determined from the available symbol. There is an icon before each function name in this column indicating if the function is a system function or a function from a user module. It also indicates if the corresponding library is 32-bit or 64-bit.
* Module ‒ This column lists the corresponding module of the function. It lists the module name only, without path. Full path information is visible on the tooltips of the corresponding module.

Other optional columns are displayed only if the corresponding events are profiled and those columns are selected for display. For TBP, the only available variable column is timer. Multiple views are possible for EBP or IBS.

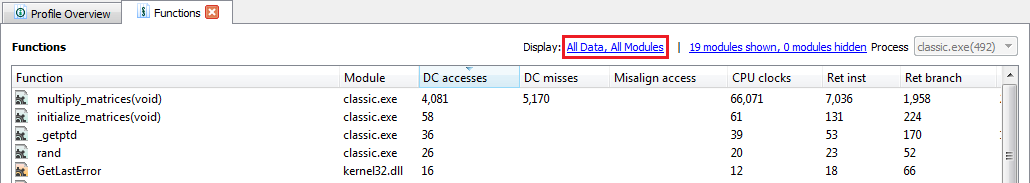
Each of the views has a corresponding list of selectable columns for display in the functions view. The combo-box at the beginning of **Column** section of **Display Settings** dialog lists the available views for the selected profile type of the current session.

Below the view combo-box is a list of checkboxes, each of which controls the visibility of a certain column. When the combo-box is checked, the corresponding column is visible; when the combo-box is unchecked, the corresponding column is hidden (see [**CodeXL CPU Profile Display Settings Dialog**](#_topic_ProfileSessionDisplaySettings)). For a full list of views and available columns, see the [**CodeXL CPU Profile Types**](#_topic_CPUProfileConfigurations) section.

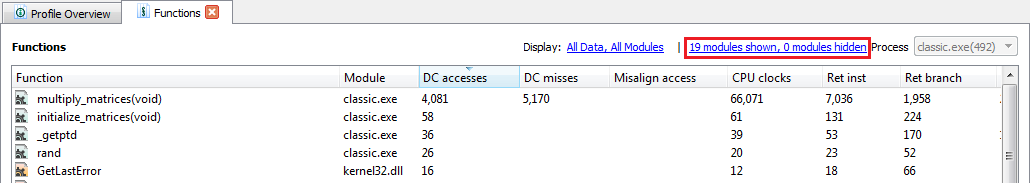
Filters

Functions table data can be filtered with one of the three options.

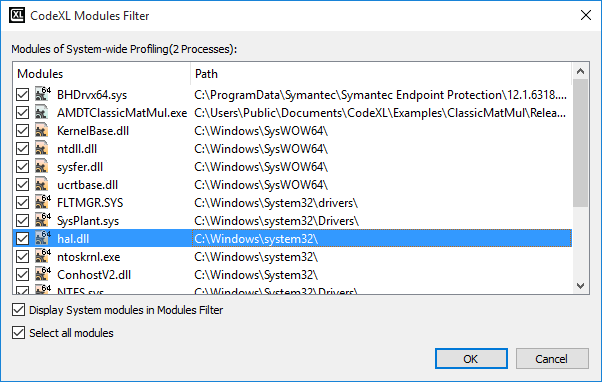
* **Display Settings dialog** ‒ This is opened from the hyperlink above the table, labeled **Display**. This dialog lets you select the view, the displayed column(s) from them. It also let you specify whether to display system modules, split the data based on core or NUMA nodes(splitting data based on NUMA nodes is available on Windows only), and display the absolute value of data or some percentage with a bar. (For details, see the [**CodeXL CPU Profile Display Settings Dialog**](#_topic_ProfileSessionDisplaySettings).)



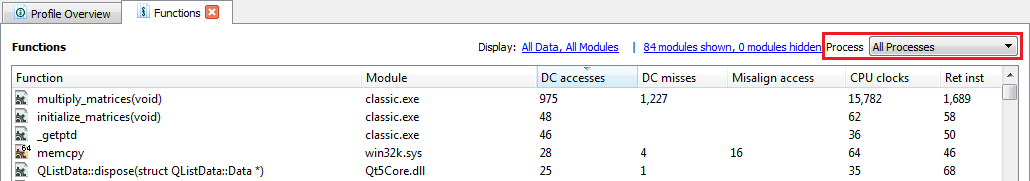
* **Module Filter** ‒ This is opened from the hyperlink above the table. It shows the number of visible and hidden modules. Clicking on this hyperlink opens the **Module Filter** dialog. It contains a table with two columns: Modules and Path. Modules column lists the modules name (the icons have the same meaning as described in the function table), and Path contains the path of that module. Both columns are sortable by clicking on the respective heading.



The following screenshot shows an example result after clicking the hyperlink shown in the red boundary above.

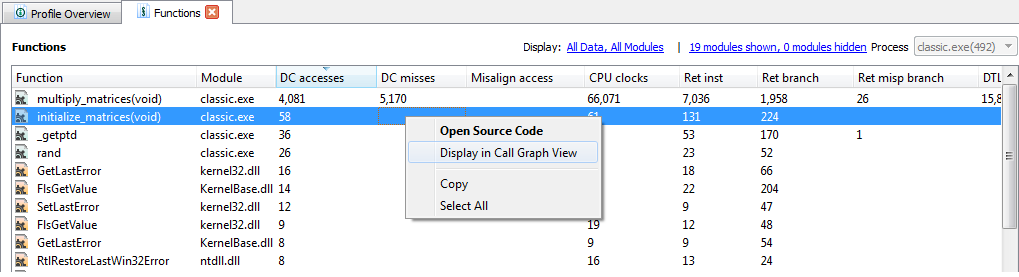


|  |  |
| --- | --- |
| **Modules** | This column contains one checkbox per cell which indicates whether to display functions of this module in functions view. There are two checkboxes below the table. **The Select all modules** checkbox is used to select/un-select all the modules currently displayed in the table. This checkbox is always enabled. The **Display System modules in Modules Filter** checkbox is enabled only if the Display system modules checkbox is checked (in the Display Settings dialog). When enabled, it shows / hides the system modules in the table. On selecting some modules in this table with the checkboxes and clicking the OK button, the functions view is updated. It shows the function only from selected modules, and the link above the table shows the updated count of shown and hidden modules. |
| **Processes** | Process combo-box lists all the processes in **process-name (process-id)** format. It also has one option called **All Processes**. If a process is selected in this combo-box, all the functions from that process are listed in the table. If **All Processes** is selected, all the functions from all the modules of the current session are displayed in the current table. |



Context Menu

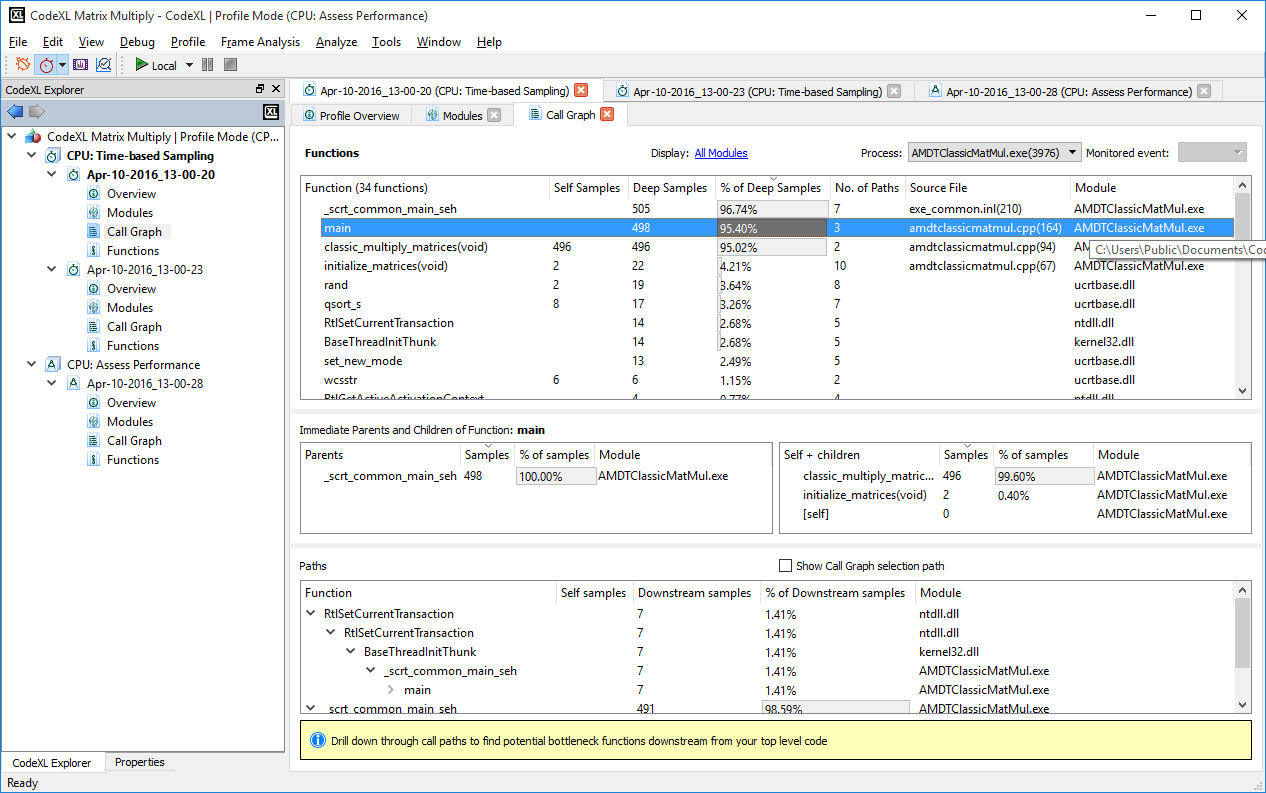
The Function table provides a context menu. Open it by right clicking on the table. The context menu has four items separated into two groups. The following list describes each of the four items.



* Opening Source Code ‒ Opens the source code or disassembly view for the currently selected function in the function table.
* Display in Call Graph View ‒ Opens the Call Graph view for the currently selected function in the function table.
* Copy ‒ Copies the selected column(s) in the function table, along with the headers of the column in buffer, for pasting into another application such as Notepad, Microsoft word, Microsoft Excel, etc. See the common context menu section.
* Select All ‒ Selects all the rows and all the columns of the function table. See the common context menu section.

#### Profile Session Call Graph View

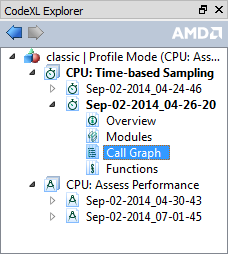
The Call Graph view displays a list of functions with their Call Graph information, including caller-to-called relationship. This can be enabled or disabled from the Call stack collection check-box of the CPU Profile Project Options for a C++ based session. Call Graph is not supported for a CLR- or Java-based session.



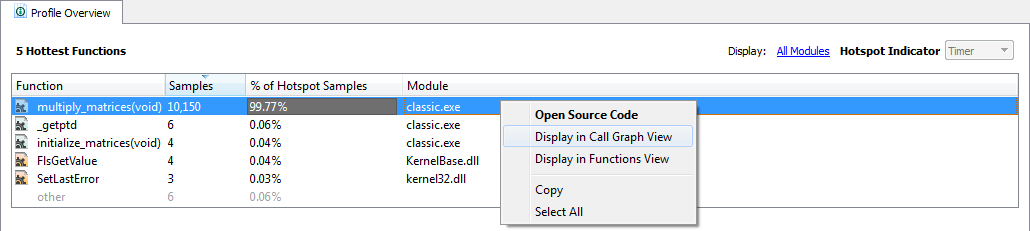
Open Call Graph View

Open this view in one of the following ways.

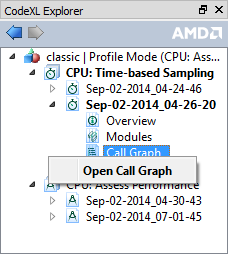
* Double click the **Call Graph** node of a session. If the Call Graph view was already there, this opens the Call Graph view tab.



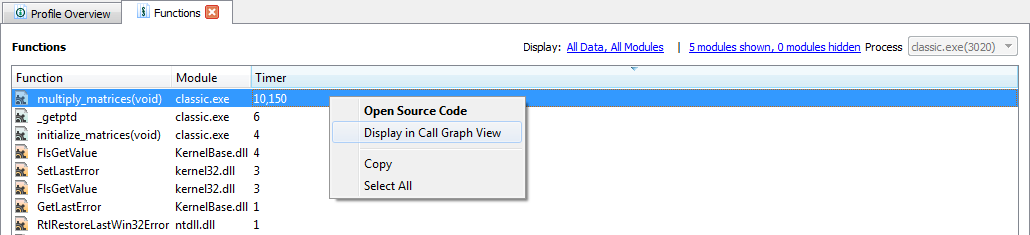
* Click the context menu item **Display in Call Graph View** of the functions table in the Profile Session Overview.



* Click the context menu item **Open Call Graph** of **Call Graph** node of a session Page. This opens the Call Graph view with that process selected. All the functions from that process are listed. This way of invoking Function view is possible only when multiple processes exist.



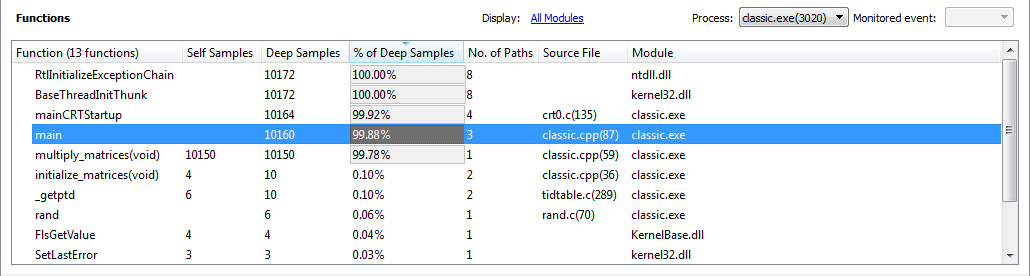
* Click the context menu item **Display in Call Graph View** of the functions table in the Profile Session Function View.



Tables in Call Graph View

There are four tables in the Call Graph View.

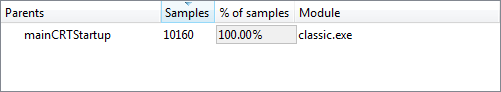
* **Function Table** ‒ Appears at the top of the page and contains the list of all functions (of the selected process or all processes if All Processes option is selected).



This table has the following columns.

|  |  |
| --- | --- |
| Function | Function name that appeared in the call stack when the chosen monitored event triggered a sample collection. Based on the configuration of **Display system modules** in the **Display Settings** dialog, this includes the function from system modules. |
| Self Samples | This shows the number of times this function was on the top of call stack when the chosen monitored event triggered a sample collection. |
| Deep Samples | Number of times this function was present in the call stack when the chosen monitored event triggered a sample collection. |
| % of Deep Samples | Percentage of deep samples out of the total count of the samples gathered **100\*(Deep Sample)/(Total Sample collected)** |
| No. of Paths | Number of unique paths containing this function. |
| Source File | The source files containing the function. The line number is shown in parentheses. |
| Module | The name of the module that contains this function. Full path is available on tooltips text of the corresponding cell of the table. |

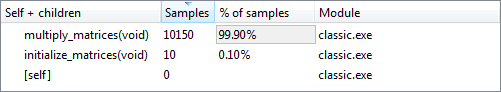
* **Parent Table** ‒ Appears on the left side of the two tables in the middle of the page. It shows the parents (the functions that call the current function) of the selected function in the function table. This table is updated when changing the selection on function table.



This table has the following columns:

|  |  |
| --- | --- |
| Parents | Lists all the function names that called the current function. In the call stack samples, those parents (direct ancestors) are immediately below the currently selected function. |
| Samples | The distribution of the currently selected function’s deep-samples between its parents. |
| % of Samples | Percentage of the paths samples from the total deep samples of the currently selected function. |

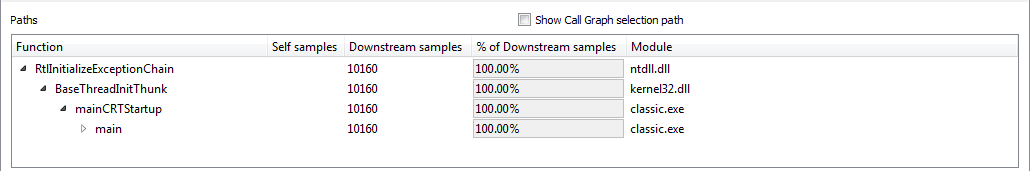
* **Children Table** ‒ Appears in the right side of two tables in the middle of the page. It shows the Children (the functions called by the current function) of the selected function in the function table. This table is updated on changing the selection on function table.



This table has the following columns:

|  |  |
| --- | --- |
| Self + children | Lists all of the function names called by the current function. In the call stack samples, those children (direct descendants) are immediately above the current function. It also lists one entry called **[self]**. This entry is on the selected function in the function table. |
| Samples | The distribution of the currently selected function’s deep-samples between its children. For the **[self]** function it is the self-samples (as there are no descendants to the function when it has been sampled itself). |
| % of Samples | Percentage of the paths samples from the total deep samples of the currently selected function. |

* **Path Graph Table** ‒ Represents the call chain in the form of a tree. In the tree, a child node represents a function called by the function represented by parent node.

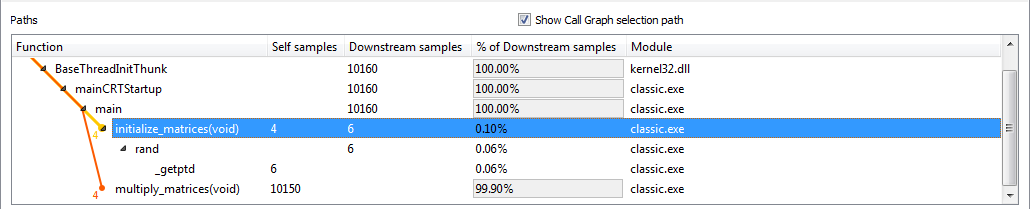


This table has the following columns:

|  |  |
| --- | --- |
| Function | The names of functions represented in the call chain tree. |
| Self samples | Number of samples in this function. |
| Downstream-samples | From this point down in the path, not including the function samples. |
| % of Downstream-samples | Percentage of downstream-samples out of all samples for the specific call path. |

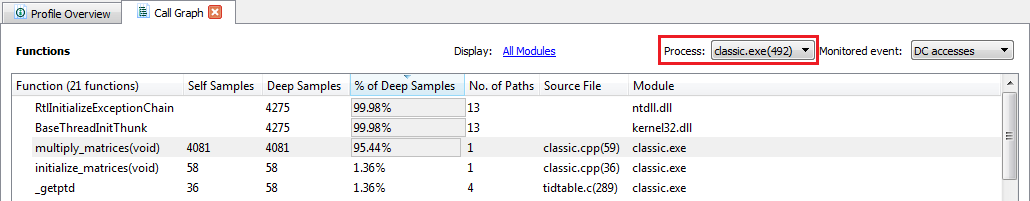
* + **Show Call Graph selection path** – This checkbox controls whether to show a line that connects all the functions in the sub-tree that form a unique path, ending in the selected function. The small number at the end of the path’s line is the length of the path (starting with 0).

The path of the selected function is colored in yellow, while the path of the function on which the cursor is currently hovering above is colored in red.



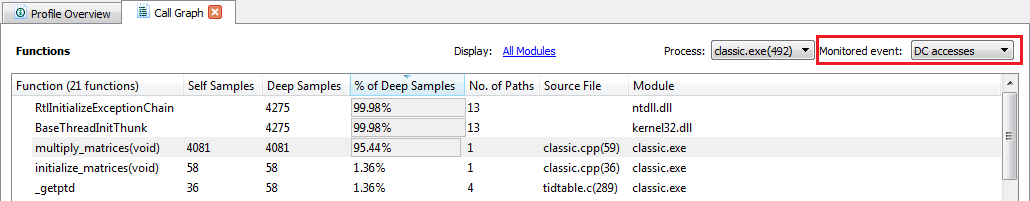
Filters

* **Processes** ‒ List all the processes in **process-name(process-id)** format



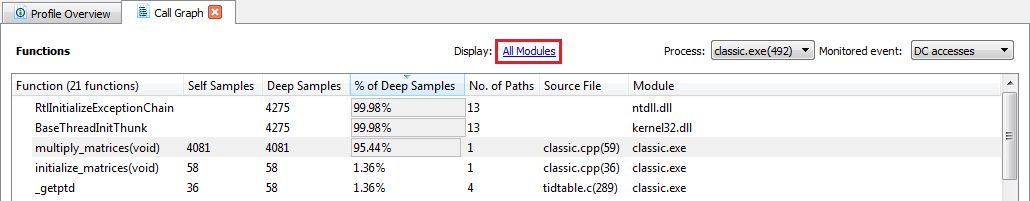
It also has the **All Processes** option. If a process is selected in this combo-box, all the functions from that process are listed in the table. If **All Processes** is selected, then all the functions from all the modules of the current session are displayed in the current table.

* **Monitored event** ‒ Combo-box at the top right



This lists the set of monitored events observed during current profiling. Only the samples, that were collected when the event selected in this combo-box was triggered, are listed.

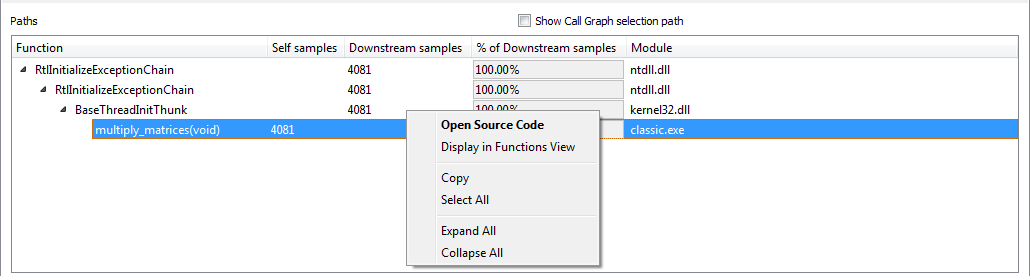
* **Display Settings dialog** ‒ Opened from the hyperlink-labeled Display above the table



Only one checkbox, **Display system modules**, is enabled when opened from **Call Graph view**, which is used to show/hide the system modules in the view.

Context Menu

The context menu provides the same information for each table discussed above, except Display Call Graph details, which are not available in the Function Table.



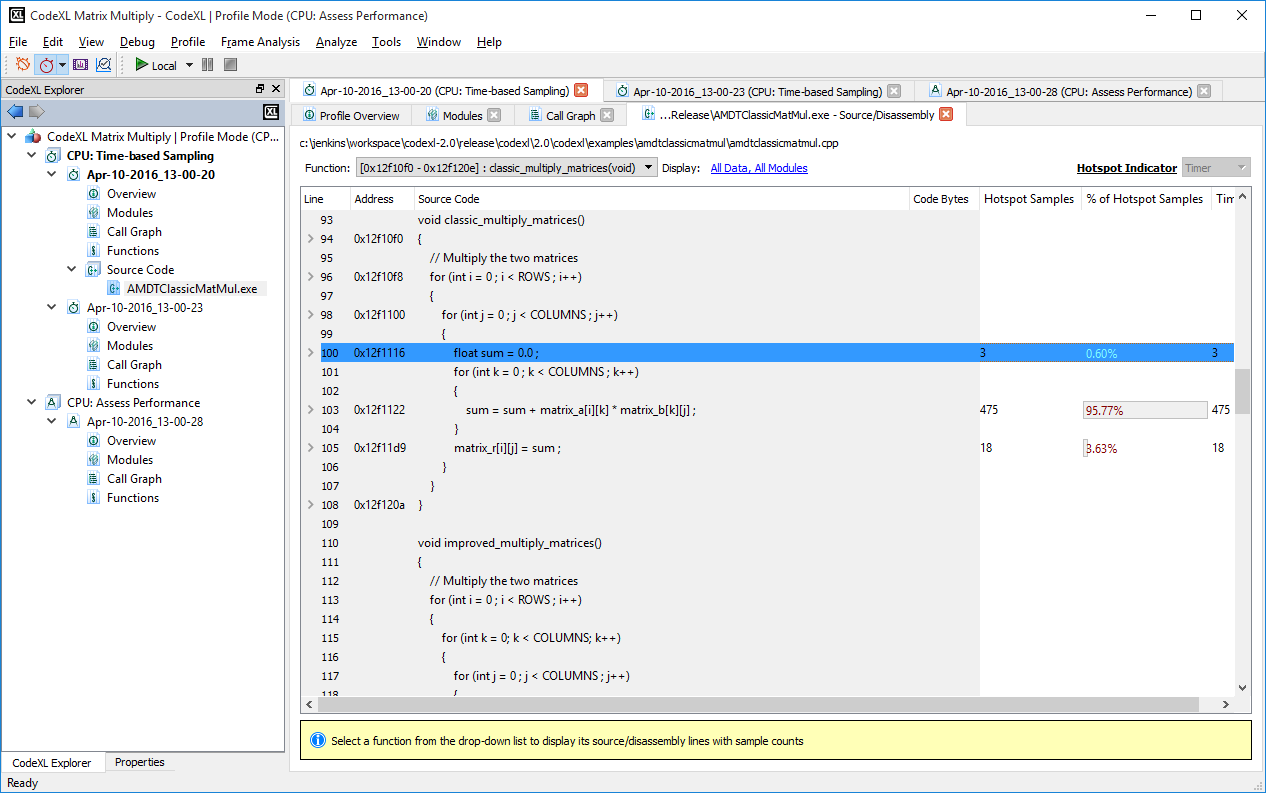
Context menu items consist of the following.

|  |  |
| --- | --- |
| **Open Source Code** | Opens the source code or disassembly view for the currently selected function in the current table. |
| **Display in Call Graph** | Opens the Call Graph view for the currently selected function in the current table. This menu item is not available for **Function table**. |
| **Display in Functions View** | Opens the Function view for the currently selected function in the current table. This menu item is not enabled for functions that are not present in **Functions View** (this can happen if the function was back-traced from another function (that may have been actually sampled), but not sampled – the monitored event was never triggered directly for that function). |
| **Copy** | Copies the selected column(s) in the current table, along with the headers to the clipboard. See the common context menu section. |
| **Expand All** | Expands all the nodes below the selected function’s node in the tree. |
| **Collapse All** | Collapses all the nodes below the selected function’s node in the tree. |

#### Profile Session Source or Disassembly View

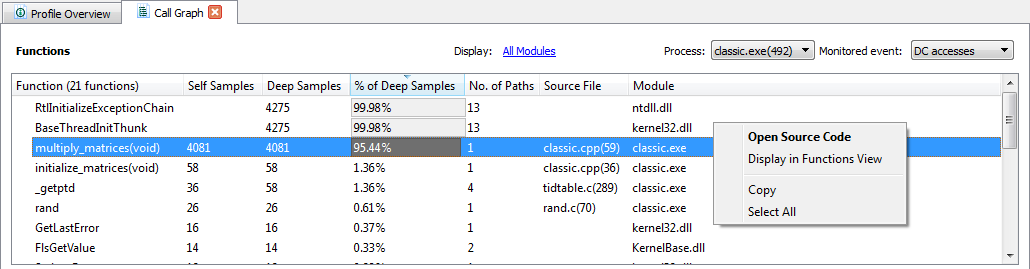
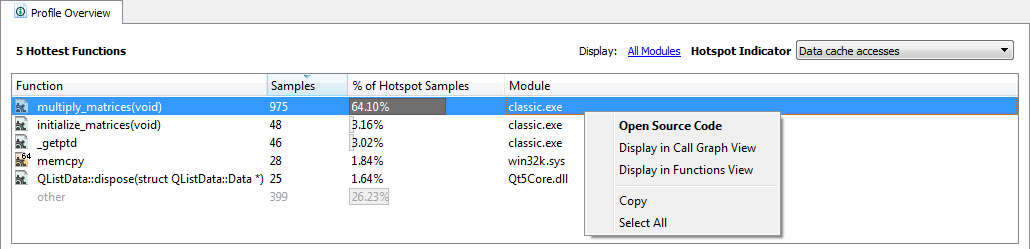
The Source Code / Disassembly view shows the source lines annotated with assembly instructions and sample count for a selected function.

Opening the Source Code View



This view can be opened from one of the following ways:

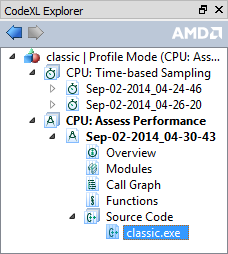
* **Overview Page**: Double-click on a function in the functions table, or right-click on a function, and select **Open Source Code**.
* **Functions View**: Double-click on a function in the functions table, or right-click on a function, and select **Open Source Code.**
* **Call-Graph View**:Right-click on a function in the functions, and select **Open Source Code**.



Open Source Code View from Overview Page Open Source Code View from Call Graph View

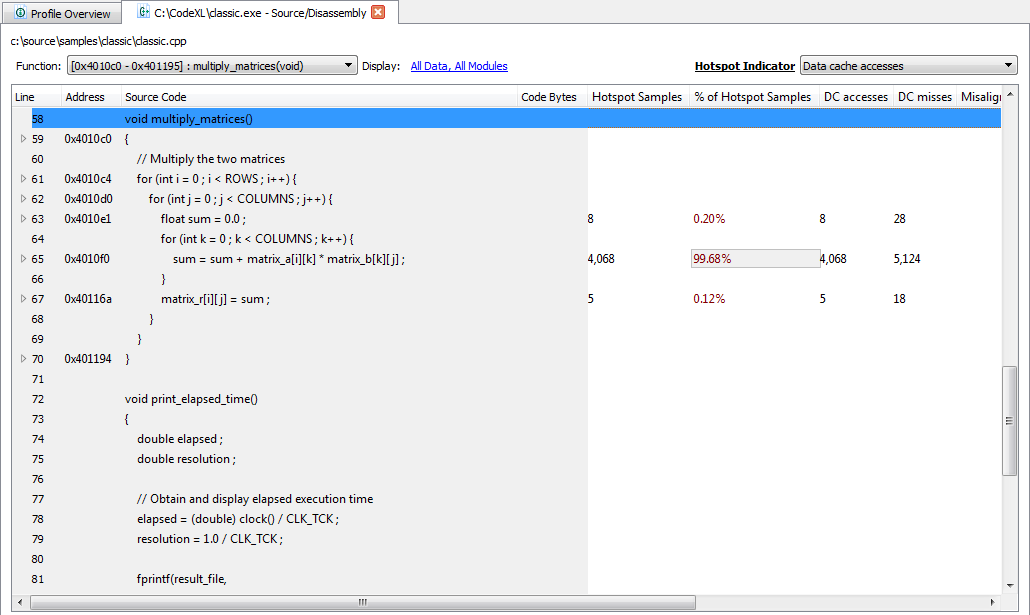
After selecting a function and clicking **Open Source Code**, a source code view will be created.

A separate instance of source code view is opened for each module. When the source code is created, a node is added in the CodeXL Explorer, called **Source Code.** Under this node, a node will be created for the module’s source code view.



If CodeXL cannot locate the source file, a file selection dialog is opened, and the user will be asked to locate the source file. If the source file will not be located, the source code view will display only disassembly.

On the top of the source code view, a combo-box is created with the list of the module’s functions. The function that was right-clicked will be selected in the functions combo-box, and will be highlighted in the code.



Source Code View opened for "multiply\_matrices" function

Data Displayed in Source Code View

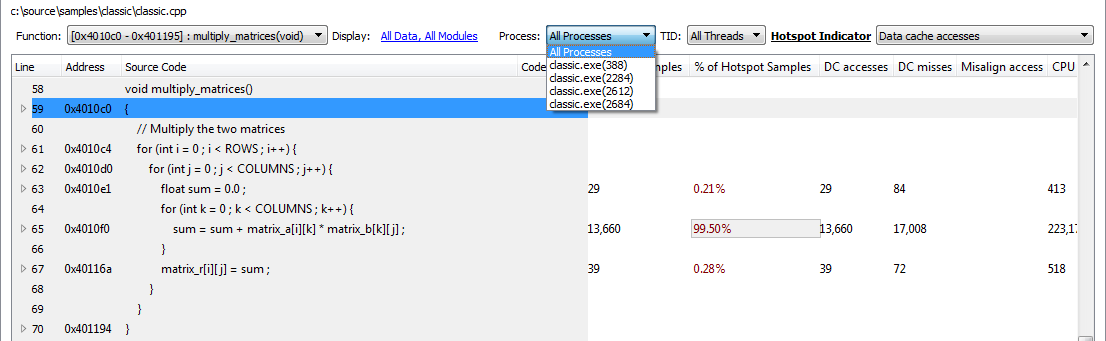
The source code consists of a table displaying the following data for each source code line / disassembly line in the current displayed source file:

|  |  |
| --- | --- |
| **Line** | The source code line number. |
| **Address** | Memory address where each instruction is located. |
| **Source Code** | The source code for the current line. |
| **Code Bytes** | Byte representations of the actual machine instructions. |
| **Hotspot Samples** | The amount of samples collected for the current hotspot indicator event / metric. The hotspot indicator is selected in the hotspot indicator combo-box. |
| **% of Hotspot Samples** | The precentage of samples collected for the current hotspot indicator event / metric. The percentage is calculated relatively to the currently display function (all the function source code lines will sum up to 100%). The hotspot indicator is selected in the hotspot indicator combo-box. |
| **Samples count for each of the collected / calculated data of the current session** | The following columns will contain the amount of samples collected for the monitored events and calculated metrics. Use the Display Settings Dialog to select which of the metrics will be displayed currently in the table. |

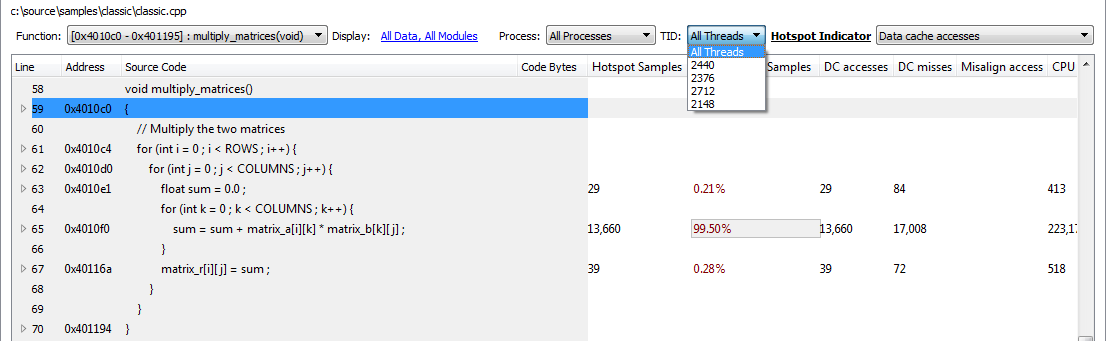
**Functions combo-box**

C:\Users\ekatz\AppData\Local\Microsoft\Windows\INetCache\Content.Word\CPU33.png

The functions combo-box contain the list of the current module’s functions. Selecting a function will highlight the function in the displayed source code, or will display disassembly code for the requested function.



Process IDs combo-box



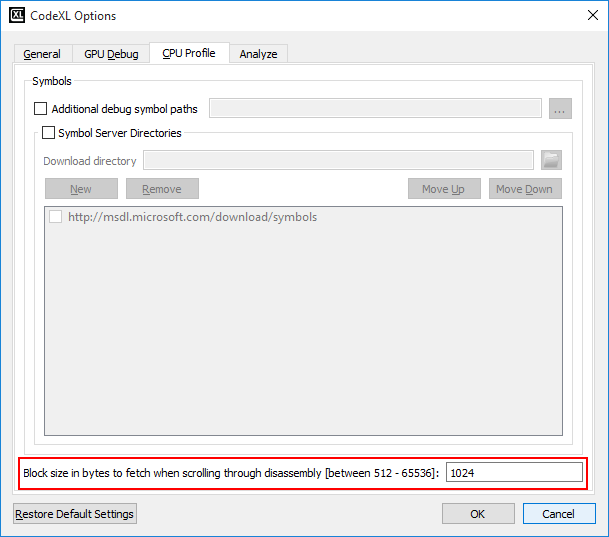
Thread IDs combo-box

**Process ID / Thread ID combo-boxes**

The process IDs / Thread IDs combo-boxes are displayed only in cases where the displayed module is multi-process or multi-threaded. Use the process IDs and thread IDs combo-boxes to display the selected process / thread ID collected samples.

Navigating through the presented data

The navigation can be done using the mouse wheel, the Page Down\Up keyboard keys or the Down\Up Arrow keyboard keys. While you navigate through a large disassembly, CodeXL will fetch another block of disassembly data as you scroll down and reach the bottom. The size of the data block that is being automatically fetched can be configured in the CPU Profiling global settings. To open the CPU Profiling global settings, go to the CodeXL menu bar and click on Tools->Options. Then, choose the CPU Profile tab, and edit the value in the text box to set the data block size in bytes:



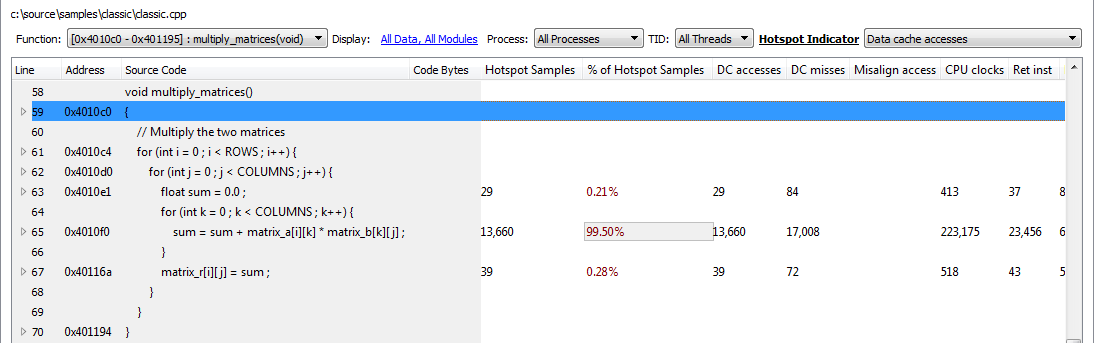
Changing the size of the data block which is being automatically fetched when scrolling through disassmbly

Source Code View Display Settings

In order to configure the source code view display settings, click the display settings link on the top of the source code view.

C:\Users\ekatz\AppData\Local\Microsoft\Windows\INetCache\Content.Word\CPU33X0.png

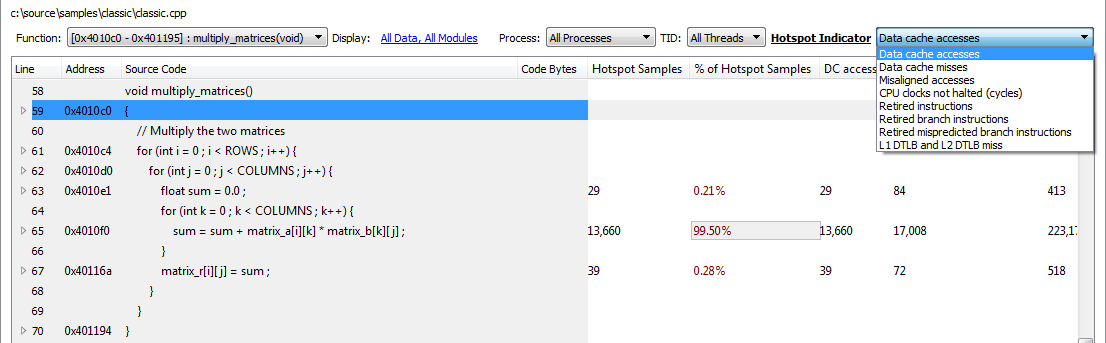
The display settings dialog can be used to select which columns will be displayed in the source code view table, to display the data separately for each core / NUMA node etc’. See more details on the display settings in Profile Session Display Settings.



The function "initialize\_matrices" is highlighted after being selected in the functions combo-box

**Hotspot Indicator combo-box**

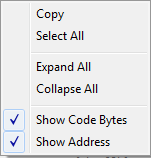
The hotspot indicator combo-box contains the list of monitored events / metrics that is currently displayed in the source code view. Each of these events / metrics can be selected. The selection of an event / metric will update the ‘**Hotspot Samples**’ and ‘**% of Hotspot Samples**’ columns to contain the data collected for this event / metric. Use the hotspot indicator combo-box to look for the most significant performance bottlenecks of your application.



Hotspot Indicator combo-box for Assess Performance Profile Session

Context Menu

The source code context menu can be opened by right-clicking on one of the items in the table.



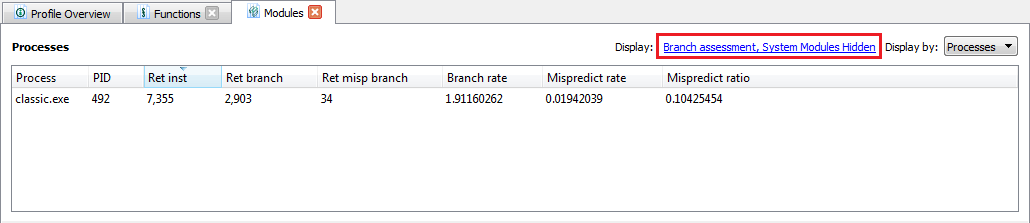
The context menu contain the following commands:

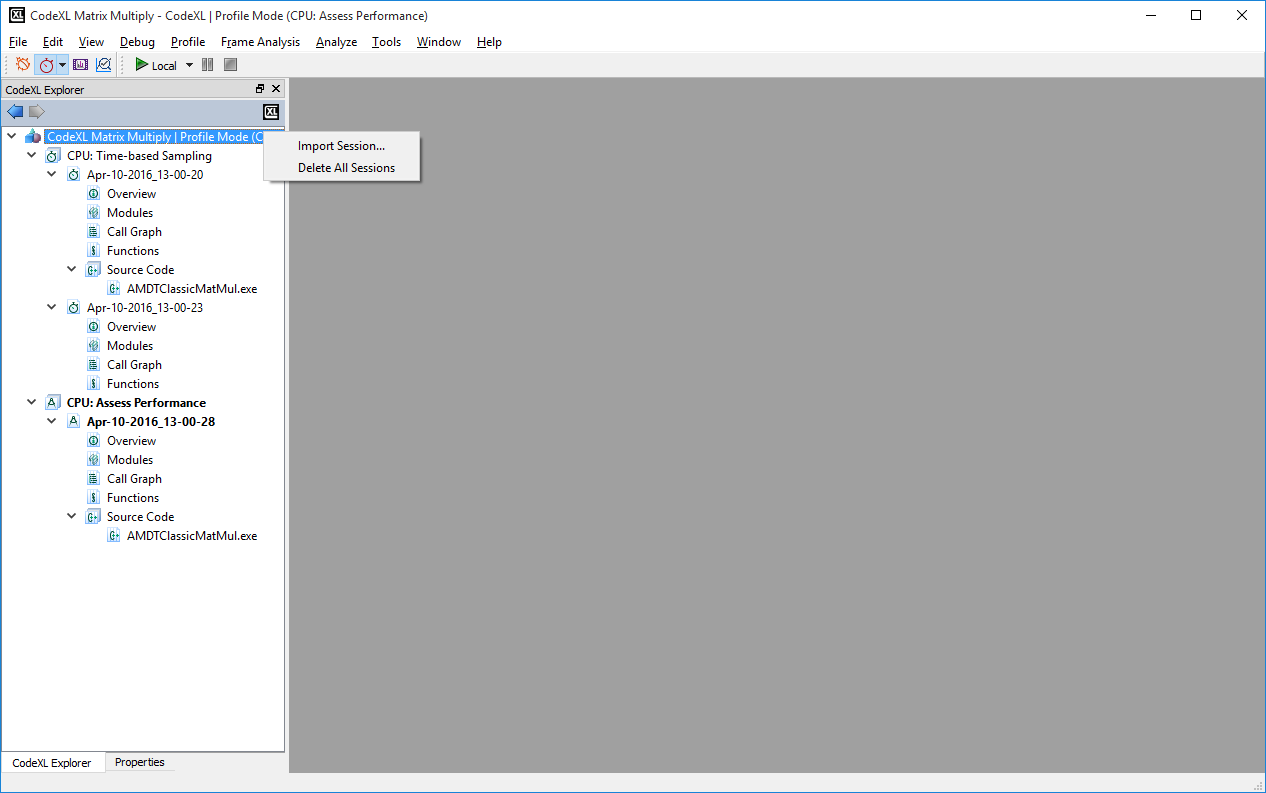
|  |  |
| --- | --- |
| **Copy** | Copy the selected row/s in the source code view table. |
| **Select All** | Select all the rows in the source code view table. |
| **Expand All** | Expand all the items in the source code table (show all the disassembly lines) |
| **Collapse All** | Collapse all the items in the source code table (hide all the disassembly lines) |
| **Show Code Bytes** | Show / hide the code bytes column in the source code view table |
| **Show Address** | Show / hide the address column in the source code view table |

#### Profile Session Display Settings

This dialog can be used to configure the display of the CPU profile for the currently opened sessions.

Click on the display link, and open the display settings dialog.





Columns

This section lets you select the group of columns displayed by the current profiled session. Each of the groups in this combo-box is a set of data columns for the current displayed session. The list of groups in this check box depends on the current session profile type.

Display System Modules

Display or hide the data collected from system modules. This option is global and affects the content of each of the opened views.

Show Percentage Bars

When this option is unchecked, the profile session data is displayed as values. When checked, the data is displayed as percentages, and percentage bars are displayed within the tables.

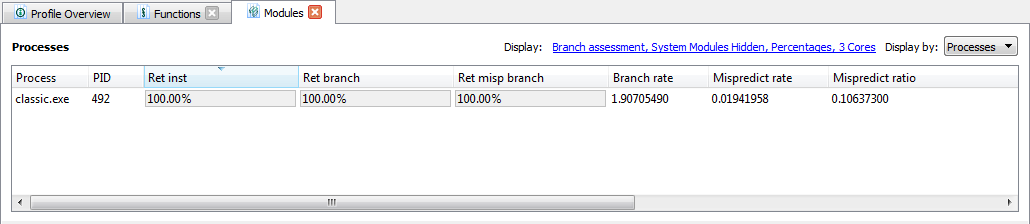
CPU Cores

Select the list of cores for which the results are to be displayed.

Separate Data Per Core / NUMA

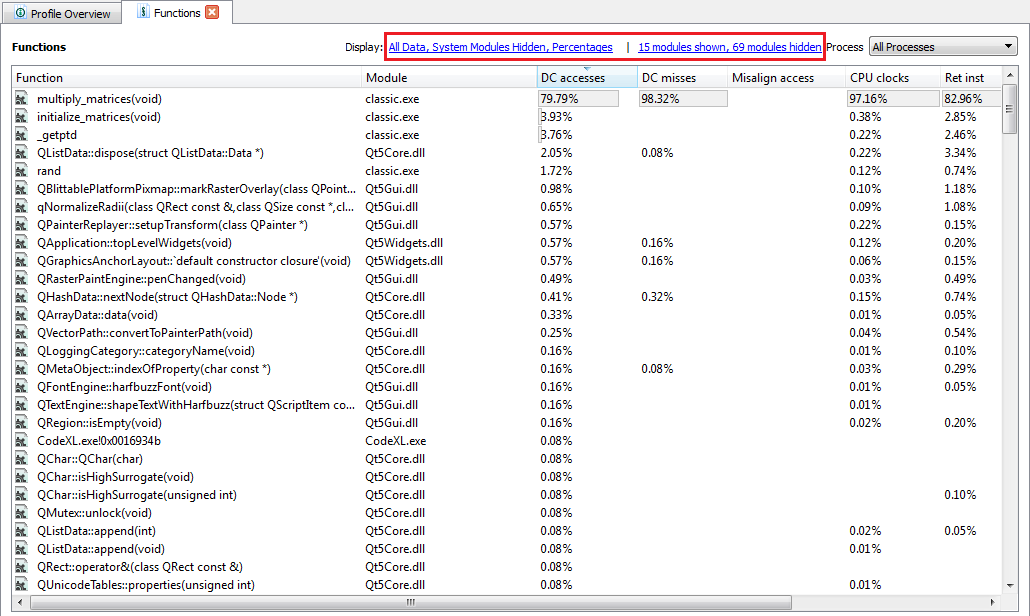
Display a separate column for each core / NUMA.

After setting the options in this dialog, click OK. CodeXL updates each of the opened sessions with the global options (percentage and system dll display); it also updates the current view with the local settings. The display filter link on the top of the views contains the current display settings. For example: The following Functions view display settings are: "Branch Assessment, System Modules Hidden, Percentages, 2 Cores" This means that the list of columns displayed for the current session are the Branch Assessment data columns. In the modules table, below the displayed columns, are the columns related to branch assessment. The data is displayed in percentages, and only two cores are selected.



Modules Filter Dialog

The below screenshot contain the display settings link for the Functions View. The right section of the string contain the amount hidden and shown modules in the functions view. Click on the right link to open the modules filter dialog.



Display System modules in Modules Filter

When this box is checked, the table above it includes the system modules. When un-checked, these modules are not shown. This option is enabled when system modules are hidden in all views.

The modules table icons represent the modules type.

|  |  |
| --- | --- |
|  | 32/64 bit user module |
|  | 32/64 bit system module |

#### Importing Profile Data

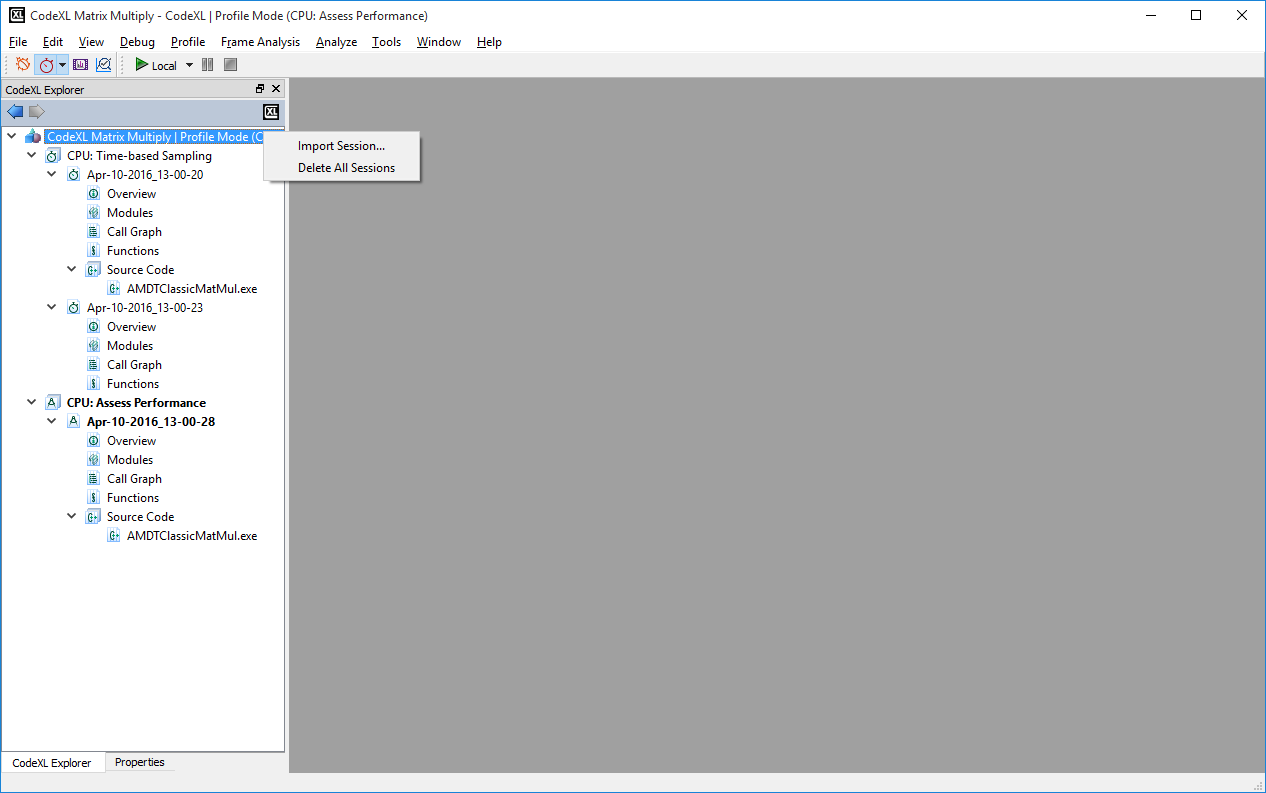
CodeXL CpuProfiler allows importing of various profile data files. The supported formats are:

1. .prd – Raw data file on Windows platform.
2. .caperf – Raw data file on Linux platform.
3. .ebp – Processed profile data file. This is an obsolete format.
4. .cxlcpdb – Processed profile data file.

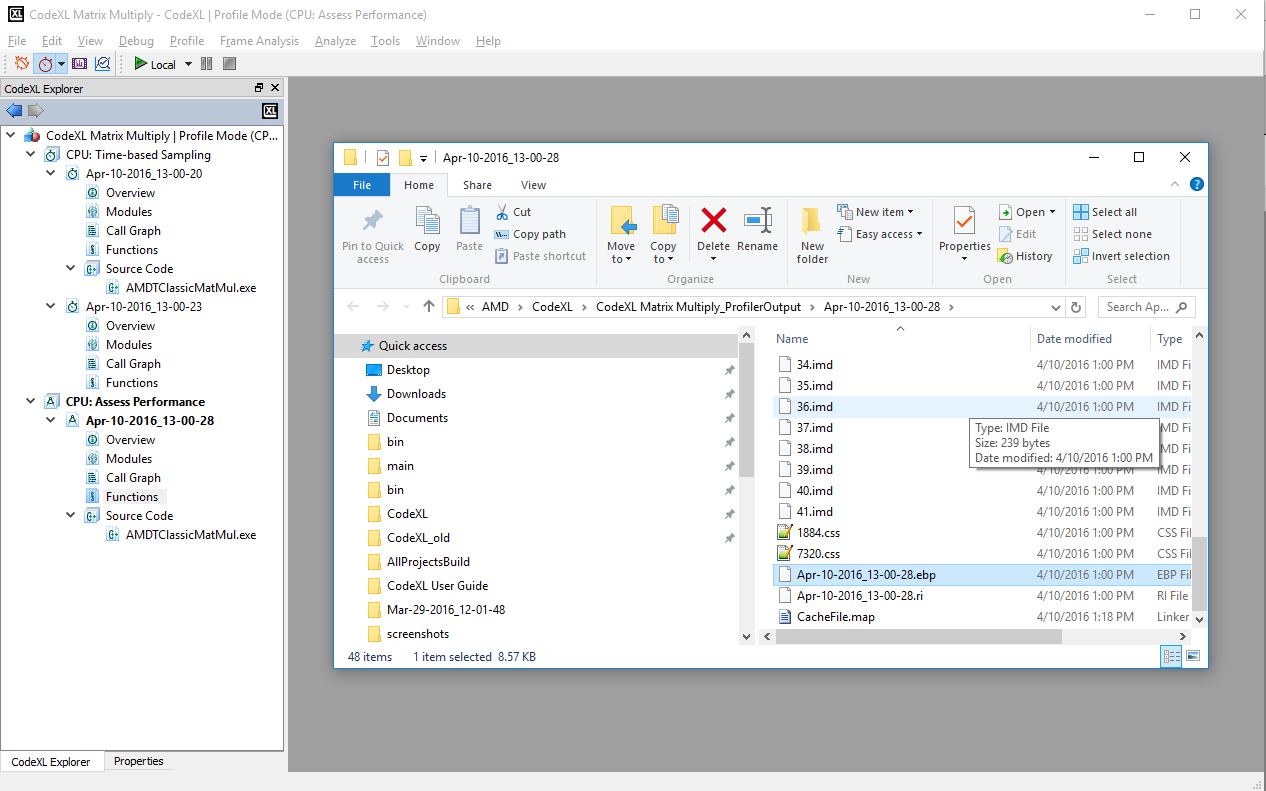
Above file formats can be imported into a CodeXL project. A new session is created for the imported profile data.

To import profile data:

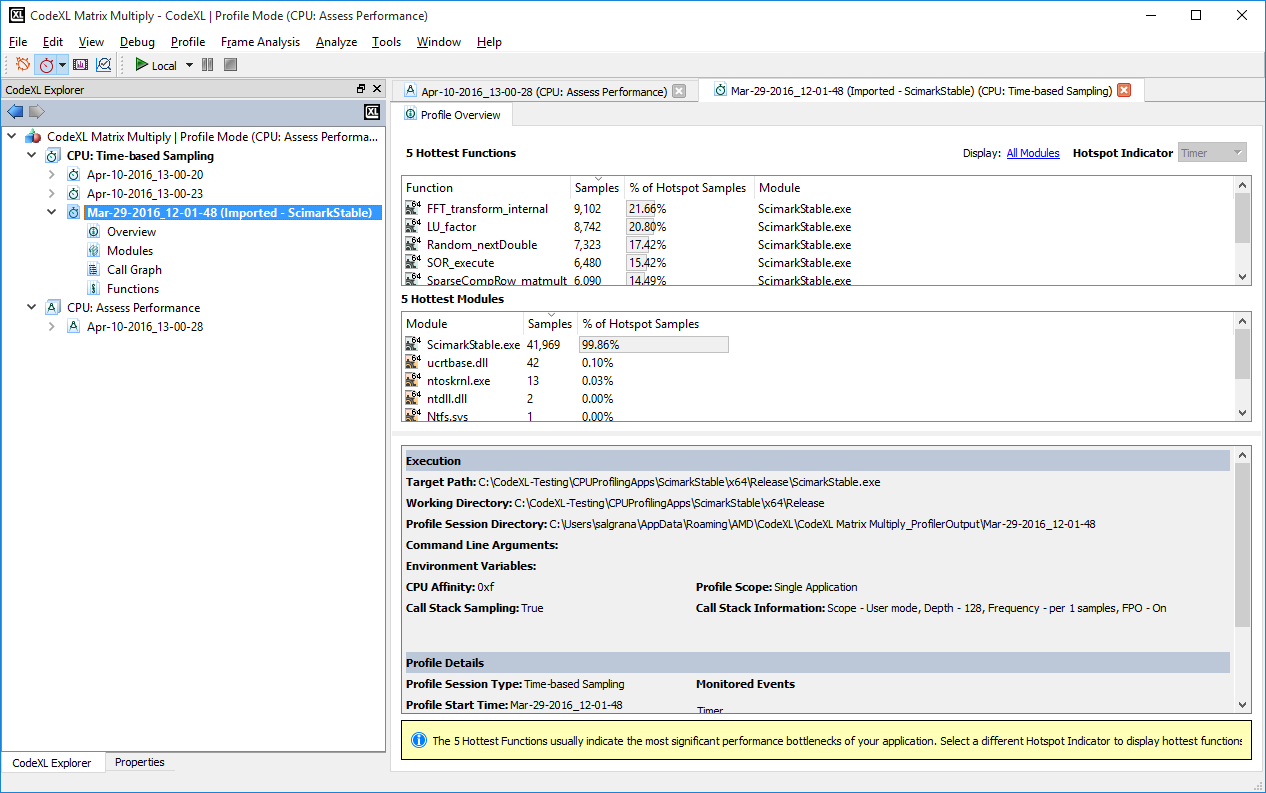
1. If a CodeXL project is not already opened, create a new CodeXL project or open an existing CodeXL project.
2. Right-click the project name in the CodeXL Explorer pane. The shortcut menu is displayed.



1. Click **Import Session...** .
2. Select any of the CodeXL .ebp/.prd/.caperf/.cxlcpdb file to be imported into the project.



1. A new session is created and displayed for the imported profile data.

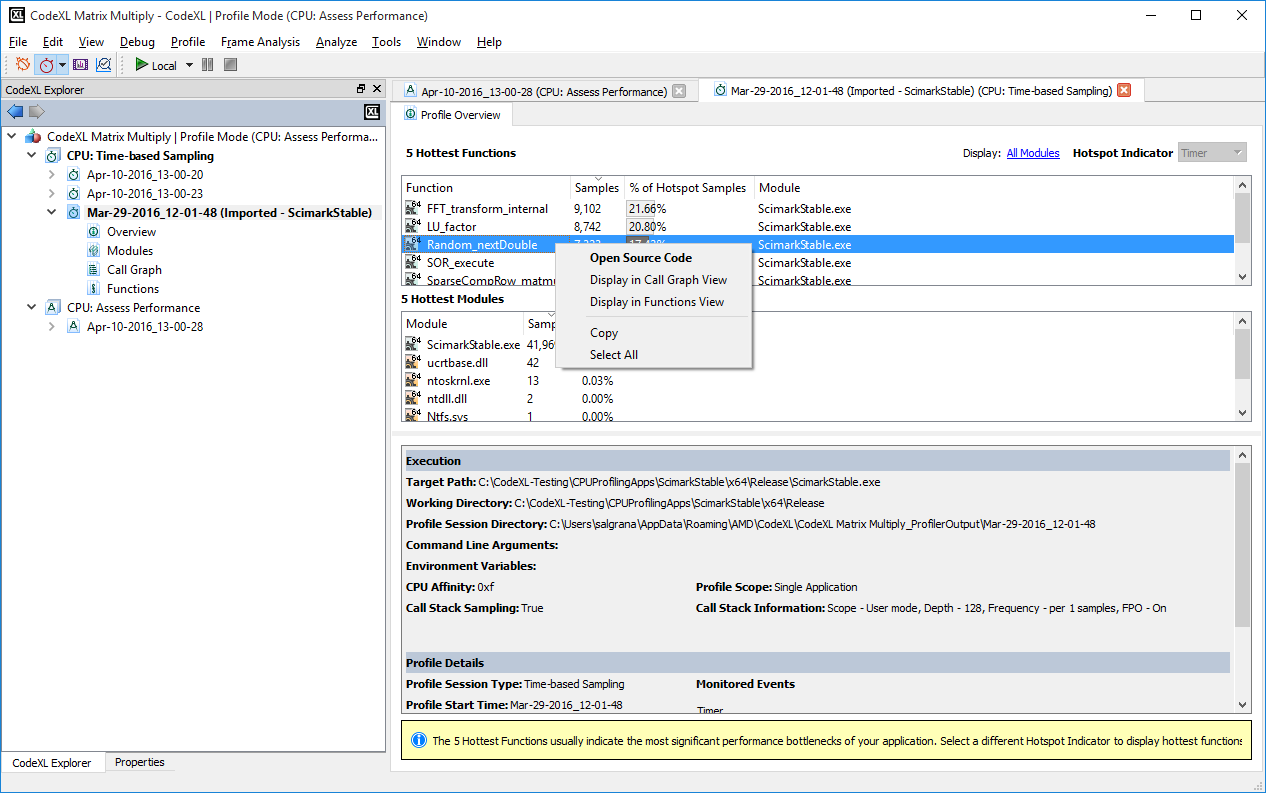


#### Saving Profile Data

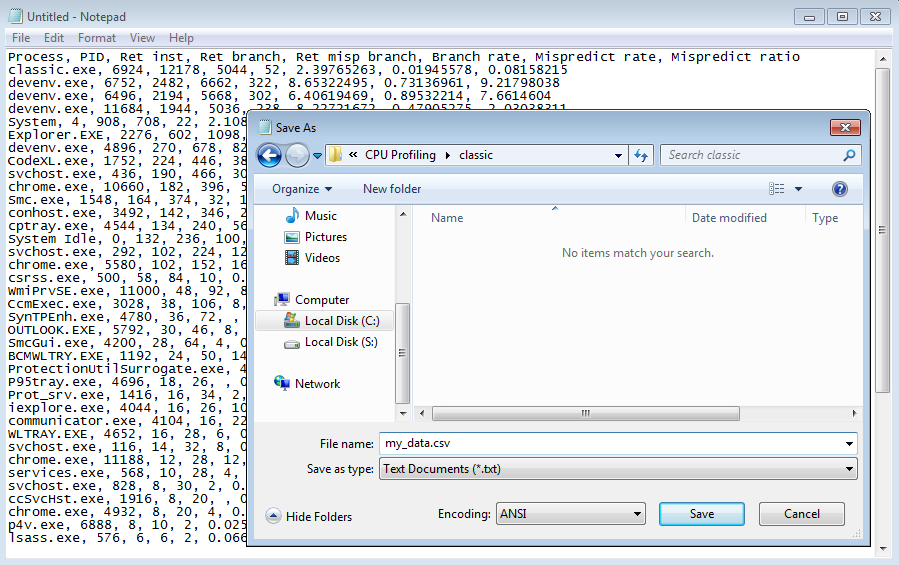
You can save the profile data in the tables of different profile session views for later analysis. This data is save in the CSV files.

To save profile data:

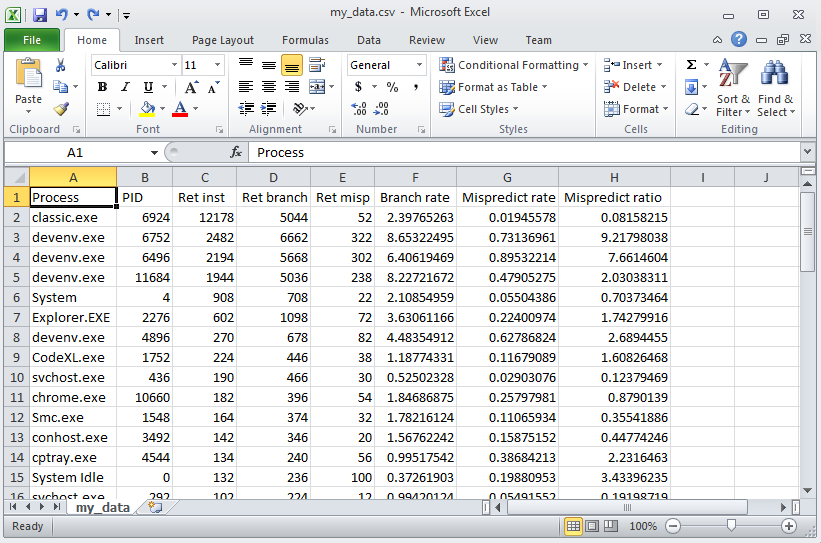
1. Right click the table having the profile data to be saved. A context-menu is displayed.
2. Click **‘Select All’**
3. Right-click and click ‘**Copy**’



1. Open notepad, and paste the copied profile data.
2. Save the file as a CSV file (with .csv extension).



1. To view the profile data, open the CSV file in a a spreadsheet program, such as Microsoft Excel or OpenOffice.org Calc.



### CPU Profile Command Line Interface

CodeXL CPU Profiler provides a command line interface utility for users who prefer to use command interpreters like **cmd.exe** on Windows and **bash** on Linux. This CLI utility will be used to collect and analyze the profile data. It can also be used from a batch file or a test script.

**Usage**: CodeXLCpuProfiler.exe command <options> [<InputApplication>] [<InputApplication’s command line arguments>]

Following commands are supported:

|  |  |
| --- | --- |
| collect | Run the given input application and collect cpu profile samples. |
| report | Process the given cpu profile data file and generate a cpu profile report in CSV format. |

Following options are supported with **collect** command.

|  |  |
| --- | --- |
| -m <profile type> | Predefined profile type to be used to collect samples. Supported profile types are:  **tbp**: Time-based Sampling  **assess**: Assess Performance  **branch**: Investigate Branching  **data\_access**: Investigate Data Access  **ibs**: Instruction-based Sampling  **inst\_access:** Investigate Instruction Access  **l2\_access:** Investigate L2 Cache Access  **clu:** Cache Line Utilization (Windows only) |
| -T <n> | Sampling interval for custom time-based profiling (tbp). Sampling interval <n> is specified in milliseconds. |
| -d <n> | Profile duration in seconds. |
| -o <file name> | Base name of the output file. If this option is specified, default path will be used. The default path will be %Temp%\Codexl-CpuProfile-<timestamp> on Windows and /tmp/Codexl-CpuProfile-<timestamp> on Linux. |
| -p <PID,PID,..> | Profile existing processes (processes to attach to). Process IDs are separated by comma. |
| -a | System Wide Profile (SWP). If this flag is not set then the command line tool will profile only the launched application or the PIDs attached. |
| -G | Enable callstack sampling with default Callstack collection sampling interval and Unwind Depth. The default values are:  **Callstack Collection Sampling Interval**: 1  **Unwind Depth**: 128 |
| -g <Interval:Depth:Scope:Fpo> | Enable callstack sampling with user defined callstack collection sampling interval, unwind depth, scope and FPO values. Scope and FPO are Windows only options.  (**Windows only**)  Scope should contain one of these options:  **user**: Collect callstacks only for code executed in user space.  **kernel**: Collect callstacks only for code executed in kernel space.  **all**: Collect callstacks for code executed in user space and kernel space.  To collect missing frames due to omission of frame pointers by compiler.  **fpo**: Collect missing callstack frames.  **nofpo**: Do not collect missing callstack frames. (default) |
| -c | Core Affinity Mask. Default affinity is all the available cores. In System-wide profiling, samples are collected only from these cores. In Per-Process profile, processor affinity is set for the launched application. |
| -f | Profile the children of the launched application (i.e. processes launched by the profiled application). |
| -b | Terminate the launched application after profile collection. |
| -s <n> | Start profiling after the specified delay duration <n> in seconds.  If n is 0, then wait indefinitely. This option is expected to be used only when the launched application is instrumented to control the profile data collection using the enable/disable APIs defined in CXLActivityLogger library, which is provided along with CodeXL installer. |
| -v | Print version string. |
| -w | Specify the working directory. Default will be the path of the launched application. |
| -h | Displays this help information. |
| -C <Custom profile> | Path to the custom profile XML file. |

Following options are supported with **report** command.

|  |  |
| --- | --- |
| -i <file name> | Input file name. Either the raw profile data file (.**prd** on Windows and .**caperf** on Linux) or the processed data file (**.ebp or .cxlcpdb**) can be specified. |
| -o <output dir> | Output directory in which the processed data file .cxlcpdb will be created. The default path will be %Temp%\<base-name-of-input-file> on Windows and /tmp/<base-name-of-input-file> on Linux. |
| -V <view xml> | Specify the View configuration XML file. All the raw data will be reported. |
| -R <section,..> | Specify the report sections to be generated. Supported report sections are:  **all**: Report all the sections.  **overview**: Report Overview section.  **process**: Report process details.  **module**: Report module details.  **callgraph:** Report callgraph.  Options **process** and **module** are mutually exclusive. Options **module** and **callgraph** are mutually exclusive. |
| -e | Specify the event index for which callgraph will be generated. This event is also used to find the hot functions in the Overview section. |
| -I | Ignore samples from System Modules. |
| -P | Show Percentage. |

**Examples**

* Launch the application classic.exe and collect Time-based profile (TBP) samples:

CodeXLCpuProfiler.exe collect -m tbp -o c:\Temp\cpuprof-tbp classic.exe

* Launch the application classic.exe and collect **assess performance** profile samples for the duration of 10 seconds:

CodeXLCpuProfiler.exe collect -m assess -o c:\Temp\cpuprof-assess -d 10 classic

* Launch the application classic.exe and collect Instruction Based Sampling (IBS) samples in **System wide profile (SWP)** mode:

CodeXLCpuProfiler.exe collect -m ibs -a -o c:\Temp\cpuprof-ibs-swp classic

* Collect Time-based profile samples in System wide profile mode for the duration of 10 seconds:

CodeXLCpuProfiler.exe collect -m tbp -a -o c:\Temp\cpuprof-TBP-swp -d 10

* Launch the application classic.exe and collect Time-based profile (TBP) samples. Also enable collecting callstack samples whenever the TBP samples are collected:

CodeXLCpuProfiler.exe collect -m tbp -G -o c:\Temp\cpuprof-tbp classic.exe

* Print help:

CodeXLCpuProfiler.exe -v

* Print version string:

CodeXLCpuProfiler.exe -h

* Once the raw cpu profile data file is generated, CodeXLCpuProfiler report command can be used to generate CSV report from that raw data file:

CodeXLCpuProfiler.exe report -i c:\Temp\cpuprof-tbp.prd -o c:\Temp\cpuprof-tbp-out

Once the raw CPU profile data file is generated using the command line utility, the results can be viewed within CodeXL using the **Import Session** command in the **CodeXL Explorer.**

#### Profile Configuration File Format

This section describes the XML configuration file passed to –C option. This data collection configuration file describes how CodeXL CPU Profiler is to be configured for data collection. Pre-defined configurations are provided with CodeXL. Advanced users can create their own data collection configuration by writing an XML file. A data configuration XML file contains only one configuration.

The <dc\_configuration> and </dc\_configuration> tags mark the beginning and end of configuration information within a data collection configuration file.

<dc\_configuration>

…

</dc\_configuration>

A collection configuration contains <tbp>, <ebp>, <ibs> or <clu> elements. Each element describes a data collection configuration of type indicated by its element name. A collection configuration should contain one or more non duplicate elements.

##### TBP Collection Configuration

The <tbp> and </tbp> tags mark the beginning and end of a time-based profile data collection configuration. This element has the following attributes:

|  |  |
| --- | --- |
| name | Configuration name (string) |
| interval | Sampling interval in milliseconds (float) |

Sample TBP configuration:

<dc\_configuration>

<tbp name=”time based profile” interval=”10.0”>

<tool\_tip> Find program hotspots </tool\_tip>

<description> Configuration to identify where an application is spending its time

</description>

</tbp>

</dc\_configuration>

##### EBP Collection Configuration

The <ebp> and </ebp> tags mark the beginning and end of an event-based profile data collection configuration. This element has the following attributes:

|  |  |
| --- | --- |
| name | Configuration name (string) |

The sampling events are specified using <event> element. One are more events can be specified. The tag <event> </event> mark the beginning and end of an event element. It describes how an individual event counter is configured for data collection.

An event has the following attributes:

|  |  |
| --- | --- |
| select | Event select value (integer) |
| mask | Unit mask value (integer) |
| os | Enables OS sampling (Boolean) |
| user | Enables user level sampling (Boolean) |
| count | Sampling period (integer) |
| edge\_detect | Enables edge detect when counting events (Boolean). This is optional. |
| host | Enables host mode event counting (Boolean). This is optional. |
| guest | Enables guest mode event counting (Boolean). This is optional. |

The values must be validated against the events and specific capabilities supported by the measurement platform. The maximum number of events depends upon the number of counters supported by the platform on which the measurements are taken.

Sample EBP configuration:

<dc\_configuration>

<ebp name=”event based profile”>

<event select=”C0” mask=”00” os=”T” user=”T” count=”250000> </event>

<event select=”76” mask=”00” os=”T” user=”T” count=”250000> </event>

<tool\_tip> HW PMC events based profiling </tool\_tip>

<description> Configuration to find potential issues for investigation

</description>

</ebp>

</dc\_configuration>

##### IBS Collection Configuration

The <ibs> and </ibs> tags mark the beginning and end of an instruction-based sampling (IBS) data collection configuration. This element has the following attributes:

|  |  |
| --- | --- |
| name | Configuration name (string) |
| fetch\_sampling | Enables IBS fetch sampling (Boolean) |
| fetch\_max\_count | Maximum periodic fetch count/sampling period (integer) |
| op\_sampling | Enables IBS Op sampling (Boolean) |
| op\_max\_count | Maximum periodic op count/sampling period (integer) |
| op\_cycle\_count | Count clock cycles (Boolean). |

Sample IBS configuration:

<dc\_configuration>

<ibs name=”instruction based sampling”

fetch\_sampling=”T”

op\_sampling=”T”

fetch\_max\_count=”250000”

op\_max\_count=”250000”

<tool\_tip> Collect data using IBS </tool\_tip>

<description> Configuration to attribute samples to instructions precisely

</description>

</ibs>

</dc\_configuration>

##### CLU Collection Configuration

The <clu> and </clu> tags mark the beginning and end of a Cache line utilization (CLU) data collection configuration. This element has the following attributes:

|  |  |
| --- | --- |
| name | Configuration name (string) |
| clu\_sampling | Enables CLU sampling (Boolean) |
| clu\_max\_count | Maximum sampling period (integer) |

Sample CLU configuration:

<dc\_configuration>

<clu name=”cache line utilization”

clu\_sampling=”T”

clu\_max\_count=”250000”

<tool\_tip> measure of cache line utilization of L1 data cache </tool\_tip>

<description> Configuration to find potential issues related to data locality and

data access pattern.

</description>

</clu>

</dc\_configuration>

##### Miscellaneous tags

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. It is usually only a few key words with no line breaks.

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

The valid values for a Boolean attribute is one of the strings “T” or “F” which denotes TRUE and FALSE respectively. The default value for Boolean attribute is “F”

### CPU Profile C/C++ Inline Functions

CodeXL CPU Profiler reports functions that are inlined and attributes samples which belong to the code region of the instances of inlined functions. In the CodeXL session views the inlined functions are displayed with the word “[inlined]” prefixed to the function name. CPU Profiler can identify the inlined functions only if the target application binary includes the information about the instances of inlined functions. If the target application binary does not contain the information about the inlined functions, then CodeXL CPU Profiler would attribute the samples to the non-inlined caller function.

Overview, Functions View, Call Graph View and Source Code view display information about inlined functions.

### CPU Profile PLT Relocations

On Linux platforms, CodeXL CPU Profiler attributes and reports the samples that belong to Procedure Linkage Table (PLT) section. PLT information is generated by compiler, which is used by dynamic linker/loader to link the application with its dependent dynamic libraries. Samples that are attributed to PLT instructions are reported against “[PLT] <function-name>” symbol. For example, an application calling a library function rand() will have an corresponding entry for this function in the PLT section. Samples attributed to this PLT entry will be shown as “[PLT] rand” and samples due to actual rand() function will be attributed to “rand()” function entry in the session views.

### CPU Profile on Virtual Machine

#### VMware Workstation

CPU Profiler supports TBP and EBP on Guest OS running on VMware Workstation 11.0 or later. It is always recommended to use latest version of VMware Workstation. Recent AMD Carrizo processor is not yet supported by VMware Workstation 11.1.x.

To run TBP within guest OS, no additional configuration needed in host OS or guest OS.

To run EBP within guest OS, please ensure the following settings are done:

* Enable Virtualization or SVM (AMD-V) in BIOS settings before booting the host OS.
* Enable AMD-V in guest OS VM settings. Edit virtual machine settings > Hardware > Processors> Virtualization engine > Enable “Virtualize AMD-V/RVI”
* Enable vPMC in guest OS VM settings. Edit virtual machine settings > Hardware > Processors> Virtualization engine > Enable “Virtualize CPU performance counters”

**Known Issues on Windows 7 Host OS:**

* When CPU Profiler EBP is running on Windows 7 host OS and a Linux guest OS is launched, crash is observed on Windows 7 due to VMware driver.
* If EBP is performed on Windows 7 host OS and EBP is performed on Linux guest OS simultaneously, then crash is observed on Windows 7 due to VMware driver.

These scenarios work fine when host OS is Windows 8, 8.1 and 10.

#### Microsoft Hyper-V

CPU Profiler supports TBP on Windows Host OS, Windows/Linux Guest OS running on Hyper-V.

CPU Profiler supports EBP only on Windows 10 Host and Windows 10 Guest OS (running on Windows 10 Host OS). Please enable Virtualization or SVM (AMD-V) in BIOS settings before booting the host OS.

#### Xen Project

CPU Profiler supports only TBP on Windows/Linux OS running on Xen hypervisor.

#### Linux KVM

CPU Profiler supports only TBP on Windows/Linux OS running on KVM hypervisor.

### CPU Profile Control APIs

CPU Profiler control APIs allow user to limit the profiling scope to a specific portion of the code within the target application. Usually, when the profiling done, it captures the samples for the complete application, i.e. start of execution till end of the application execution. The control APIs can be used to enable the profiler only for a specific part of application, e.g. a CPU intensive loop, a hot function, etc. The target application need to be recompiled after adding the control APIs within the application.

The control APIs:

// To pause CPU profiling, call one of the below two APIs.

int amdtStopProfiling(amdtProfilingControlMode); // Set mode to AMDT\_CPU\_PROFILING

int amdtStopProfilingEx(void);

// To resume CPU profiling, call one of the below two APIs.

int amdtResumeProfiling(amdtProfilingControlMode); // Set mode to AMDT\_CPU\_PROFILING

int amdtResumeProfilingEx(void);

CPU Profiler only profiles the code within each Resume, Stop APIs pair. Refer “CPU Profiler Tutorial” on how to use these APIs, compile your target application and profile only the desired part of code.

### CPU Profile IMIX report generation

If you are interested in the hot instructions for a target application then in such IMIX report will be useful. IMIX report generates report on hotspot instructions summary.

Sample IMIX report summary:

|  |  |  |
| --- | --- | --- |
| **Disassembly** | **Samples Percentage** | **Samples Count** |
| mov [rsp+08h],rcx | 5.54 | 111 |
| retnq | 3.54 | 71 |
| mov rax,[rsp+08h] | 3.09 | 62 |
| sub rsp,28h | 2.79 | 56 |
| mov [rsp+18h],r8d | 2.44 | 49 |
| mov [rsp+10h],edx | 2.34 | 47 |

Only CPU Profiler CLI interface supports IMIX report generation. Use option ‘-R imix’ in CLI during report generation to generate IMIX information. Refer CodeXLCpuProfiler help (-h) option to get more details.