Implementation Reference for Analysis Communication Agent

Version 1.0 03/05/2020

Table of Contents

1 **Introduction** 5
1.1 **Overview** 5
1.2 **Scope** 5
1.3 **Terminology** 5
1.4 **Security Disclaimer** 5
1.5 **Supported Analysis Types** 6
1.5.1 Hotspots Analysis 6
1.5.2 Microarchitecture Analysis 6
1.6 **System Requirements** 6
1.7 **License Information** 7
2 **Run a Target Collection** 7
2.1 **On the Target** 7
2.1.1 Download the Analysis Communication Agent Reference Solution 7
2.1.2 Repository layout: 7
2.1.3 Build Sampling Driver 8
2.1.4 Load Sampling Driver 8
2.1.5 Build Analysis Communication Agent 8
2.1.6 Start Analysis Communication Agent 8
2.1.7 Remove Sampling Driver 9
2.2 **On the Host** 9
2.2.1 Download and Install Intel VTune Profiler 9
2.2.2 Start Intel VTune Profiler 9
2.2.3 Configure and Run Remote Analysis 11
2.2.4 View Collected Data 12
3 Architecture 13
3.1 **Hardware PMU Counters** 13
3.2 **Sampling Target Collector Design** 13
3.2.1 Overview 13
3.2.2 Analysis Communication Agent Reference 14
4 Porting Requirements 14
4.1 **Hardware Access Mechanisms** 14
4.2 **Operating System Services** 15
5 **Collection Workflow** 16
6 **Required Data for Performance Profile** 17
6.1 Platform Topology 17
6.2 Process, Thread and Module Information 17

6.3 Performance Data 18
7 Advanced Features 18
7.1 Precise Event Based Sampling (PEBS) 18
7.2 Last Branch Record (LBR) 18
7.3 Event Multiplexing 18
7.4 Uncore Profiling....19
8 Source directory structure....19
8.1 Top level source structure....19
8.2 Source structure for agendk....19
8.3 Source structure for sepdk/include folder....20
8.4 Source structure for sepdk/src folder 20
9 Implementation Details....21
9.1 Analysis Communication Agent....21
9.1.1 Overview 21
9.1.2 Communication Protocol 22
9.1.3 Communication Channels 23
9.1.4 Porting Requirements 23
9.2 Sampling Driver 25
9.2.1 Overview 25
9.2.2 Enable Hotspot Analysis....26
9.2.3 Enable Microarchitecture Exploration analysis....28
9.2.4 Enable Memory Access Analysis 29
10 Test Utilities....31
10.1 Preparation of Target Platform....31
10.1.1 Build Test Application on Target 31
10.1.2 Execute the Test Application....31
10.1.3 Configuration of Test Session....31
10.1.4 Run Tests 32
10.2 Test Scenarios....32
10.2.1 TerminateTest 32
10.2.2 VersionTest....32
10.2.3 SetupInfoTest 32
10.2.4 SysConfigSizeTest 32
10.2.5 SysConfigTest 32
10.2.6 PlatformInfoTest 32
10.2.7 InitNumDeviceTest....32
10.2.8 BusyDriverTest 32

10.2.9 SetEventConfigTest 32
10.2.10 GetTscTest 32
10.2.11 GetNumCoresTest 33
10.2.12 CollectionTest 33
10.2.13 UncoreCollectionTest 33

1 Introduction

This document describes the implementation of Analysis Communication Agent to enable performance analysis of custom embedded operating systems (OS) or real-time operating systems (RTOS) by Intel® VTune™ Profiler. This is a reference implementation that also provides guidance on enabling a sampling collector for this purpose.

1.1 Overview

In this reference solution, the data collections are launched remotely on the target systems that run the sampling collector.

The solution has two components:

- **Analysis Communication Agent** – A software agent that runs on the remote target system. This program works as a bridge between the SEP collector (available in Intel VTune Profiler) and a sampling driver loaded into the Linux kernel. This program runs in the user mode.
- **Sampling Driver** – A module loaded in the kernel that includes the implementation to collect performance data on the target system.

1.2 Scope

This reference implementation uses a Linux operating system. However, you can implement the collector on any operating system compatible with Intel VTune Profiler for data collection and visualization.

1.3 Terminology

SEP	Sampling Enabling Product – Sampling collector that is a part of Intel VTune Profiler and runs on the host
Analysis Communication Agent	Component of the target collector that runs in user mode
Sampling Driver	Component of the target collector that runs in kernel mode
IOCTL	I/O Control – Mechanism to transfer data between user mode and kernel mode
MSR	Model Specific Register
PCI	Peripheral Component Interconnect
MMIO	Memory Mapped I/O
APIC	Advanced Programmable Interrupt Controller
PMU	Performance Monitoring Unit
PEBS	Precise Event Based Sampling
LBR	Last Branch Records

1.4 Security Disclaimer

IMPORTANT:

- **Analysis Communication Agent requires a VTune collector binary to run on the host system and communicate with an agent running on the target RTOS system via TCP/IP connection. Note that this is an unsecure TCP/IP connection. It is recommended that you use this reference solution in a secure lab environment at this time.**

- **The implementation enables read/write access mechanism to various MSR/PCI/MMIO registers. To eliminate accesses to registers beyond the performance monitoring related, the implementation includes checks for valid register addresses. The implementation specific to your RTOS must include these checks.**

1.5 Supported Analysis Types

Intel VTune Profiler supports these key performance analysis types:

1.5.1 Hotspots Analysis

Use **Hotspots Analysis** to perform these tasks:

- Analyze call paths.
- Identify locations where your code is spending the most time.
- Discover opportunities to tune your algorithms.

1.5.2 Microarchitecture Analysis

The **Microarchitecture Exploration** analysis type identifies the CPU pipeline stage (such as front-end or back-end) and hardware units responsible for hardware bottlenecks.

For more information on these analysis types, see the Intel VTune Profiler User Guide.

1.6 System Requirements

Ensure that your system meets these requirements before enabling Analysis Communication Agent.

- **TCP/IP protocol communication between host and target**
  The sampling collector inside Intel VTune Profiler running on the host must communicate with Analysis Communication Agent to send and review control and data messages. In addition, it is strongly required that the host and target are placed within the same subnet. Otherwise, a collection may fail where there is heavy data traffic between them.
- **Exclusive access to Performance Monitoring Unit (PMU)**
  The target collector uses the Performance Monitoring Unit (PMU) in hardware to collect profiling data. It requires exclusive read/write access to various MSRs for the PMU. If other programs use the PMU concurrently, this can result in an invalid output. For example, a PMU counter could be used by watchdog timer on Linux kernel. In this case, the watchdog timer must be disabled while collecting data. The complete list of MSRs are defined in the Intel® Architecture Software Developer Manual.
- **Support for socket APIs**
  Your operating system must support socket APIs for proper internal communication.
- **At least 100MB of file or memory space on target for delayed transfer mode**
  The data channels in Analysis Communication Agent support two data transfer modes - immediate transfer and delayed transfer. In the immediate transfer mode, the host receives data from the target as soon as it is generated. In the delayed transfer mode, data is stored in temporary files during data collection. The host receives the data after the collection completes. By default, Analysis Communication Agent uses the immediate transfer mode for data collection, which requires no additional file space. If minimal system perturbation is favorable during collection, the delayed transfer mode is a better choice. In this mode, the target machine needs at least 100MB of file space to save temporary data.
  Please note that the implementation for the Operating Systems with no writable file system space available may use target machine memory (DRAM) to avoid any file system operations.

1.7 License Information

Analysis Communication Agent Reference Solution (containing Analysis Communication Agent and Sampling Driver) is released under MIT License.

2 Run a Target Collection

This workflow describes how to run a target collection using Analysis Communication Agent Reference Solution and Intel VTune Profiler. The example employs a Linux target system.

Workflow:

1. On the target,
   i. Download the Analysis Communication Agent Reference Solution.
   ii. Build Sampling Driver.
   iii. Load Sampling Driver.
   iv. Build Analysis Communication Agent.
   v. Start Analysis Communication Agent.
   vi. Remove Sampling Driver.
2. On the host,
   i. Download and install VTune Profiler.
   ii. Start Intel VTune Profiler.
   iii. Configure and run remote analysis.
   iv. View collected data.

2.1 On the Target

2.1.1 Download the Analysis Communication Agent Reference Solution

Analysis Communication Agent Reference Solution is available as an open source kit here:

https://github.com/intel/aca.

The repository contains:

- Analysis Communication Agent
- Sampling Driver
- Supporting documentation

To clone the repository type:

```
$git clone https://github.com/intel/aca
```

2.1.2 Repository layout:

The repository layout is shown below. The agentdk folder contains the code for user-mode Analysis Communication Agent component. The sepdk folder contains code for the Sampling Driver kernel component. The documentation is inside the docs folder and tests directory contains sample test programs for verifying the functionality.

```
├── agentdk
├── docs
├── sepdk
├── tests
├── CONTRIBUTING.md
├── LICENSE.txt
└── README.md
```

2.1.3 Build Sampling Driver

Change the directory to sepdk folder and locate the build-driver script:

```
$ cd < install_dir >/sepdk/src
```

Use the build-driver script to build the drivers for your kernel:

```
$ ./build-driver –ni
```

When the drivers are built successfully, the build script generates `sep5.ko` which is a loadable kernel module.

To get help on using the build-driver script, type:

```
./build-driver -h
```

2.1.4 Load Sampling Driver

Install drives manually with the `insmod-sep` script:

```
$ cd < install_dir >/sepdk/src
$ ./insmod-sep -g <group>
```

`<group>` is the group of users that have access to the driver.

Ensure that the driver is properly loaded:

```
$ lsmod |grep sep
```

On some systems, Sampling Driver requires that you disable NMI watchdog.

To disable NMI watchdog during the sampling collection, use either of these commands:

```
$ echo "0" | sudo tee /proc/sys/kernel/nmi_watchdog
```

or

```
$sudo sysctl kernel.nmi_watchdog=0
```

2.1.5 Build Analysis Communication Agent

Change the directory to the agentdk folder and locate the makefile:

```
$: cd < install_dir >/agentdk/
$: make
```

This command generates the Analysis Communication Agent executable (sepagent executable).

2.1.6 Start Analysis Communication Agent

There are two requirements before starting the agent:

First, the agent and the host are required to be within the same subnetwork. Technically, the IP address with the subnet mask in both agent and host should be identical.

Second, the NMI watchdog must be disabled on target. To disable it, type:

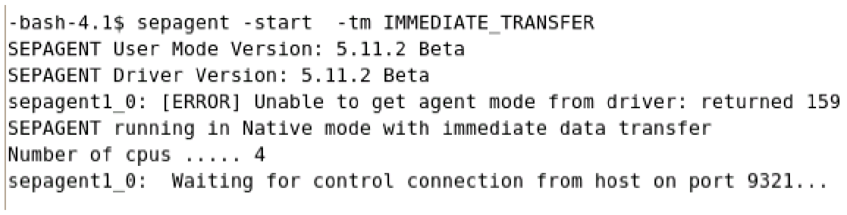
```
$: echo 0 > /proc/sys/kernel/nmi_watchdog
```

To start Analysis Communication Agent, type:

$\$: sepagent - start$

or

$\$: sepagent - start - tm\ IMMEDIATE\_TRANSFER$



```
-bash-4.1$ sepagent -start  -tm IMMEDIATE_TRANSFER
SEPAGENT User Mode Version: 5.11.2 Beta
SEPAGENT Driver Version: 5.11.2 Beta
sepagent1_0: [ERROR] Unable to get agent mode from driver: returned 159
SEPAGENT running in Native mode with immediate data transfer
Number of cpus ..... 4
sepagent1_0:  Waiting for control connection from host on port 9321...
```

`sepagent` displays a message indicating that it is waiting for a connection from the host.

For information on using `sepagent`, type:

$sepagent - help$

2.1.7 Remove Sampling Driver

To remove Sampling Driver, use the `rmmod-sep` script:

$\$\ cd\ \ < install\_dir > /sepdk/src$

$\$\ ./rmmod\text{-}sep - s$

2.2 On the Host

2.2.1 Download and Install Intel VTune Profiler

Download Intel VTune Profiler from https://software.intel.com/en-us/vtune/ . Choose the appropriate package for your OS and install it.

Launch installer GUI on Linux OS:

1) Extract the installation package to a writeable directory:

   $tar\ \ - xzf\ vtune\_amplifier\_ < version > .tar.gz$

2) Navigate to the directory containing the extracted files.
3) Run the installer:

   $./install\_GUI.sh$

Launch installer GUI on Windows OS:

Double-click the $VTune\_Profiler\_ < version > \_setup.exe$ file to start the installation.

See these links for detailed installation instructions:

Linux OS: https://software.intel.com/en-us/vtune-amplifier-install-guide-linux

Windows OS: https://software.intel.com/en-us/vtune-amplifier-install-guide-windows

2.2.2 Start Intel VTune Profiler

After setting up appropriate environment variables, you can start Intel VTune Profiler through:

- Graphical interface
- Command-line interface

2.2.2.1 Set Up Environment Variables (optional)

Run the `vtune-vars` script to set up environment variables:

Linux OS:

`bash` users: *source <install_dir>/vtune-vars.sh*

By default, the <*install_dir*> is:

For root users: $/opt/intel/vtune\_profiler\_version$

For non-root users: $\$HOME/intel/vtune\_profiler\_version$

Windows OS:

<install_dir>|vtune-vars.bat,

By default <*install_dir*> is $C{:}\backslash[Program\ Files\ (x86)]\backslash IntelSWTools\backslash$ $VTune\ Profiler\ version.$

When you run the `vtune-vars` script, it displays the product name and the build number.

2.2.2.2 Start Intel VTune Profiler from GUI

On Windows OS, launch the standalone GUI either via the Search menu or by locating the product from the Start menu.

2.2.2.3 Start Intel VTune Profiler from Command Line

If you passed step 2.2.2.1, you can now use these commands to start Intel VTune Profiler:

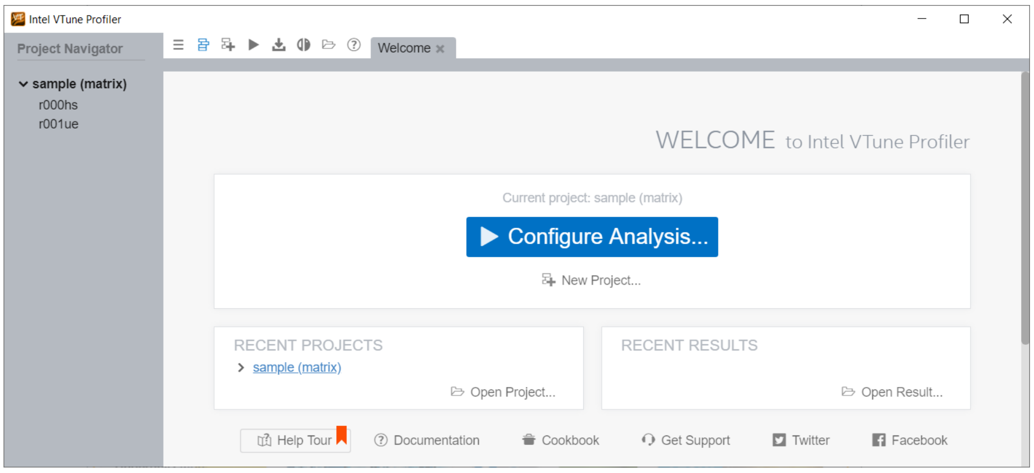
- Graphical interface : *vtune*
- Command-line interface : *vtune-gui*

Otherwise launch Intel VTune Profiler by absolute path:

- Graphical interface : $< install\_dir >/bin64/vtune$

  Command-line interface : $< install\_dir >/bin64/vtune\text{-}gui$

Intel VTune Profiler opens with this welcome screen.

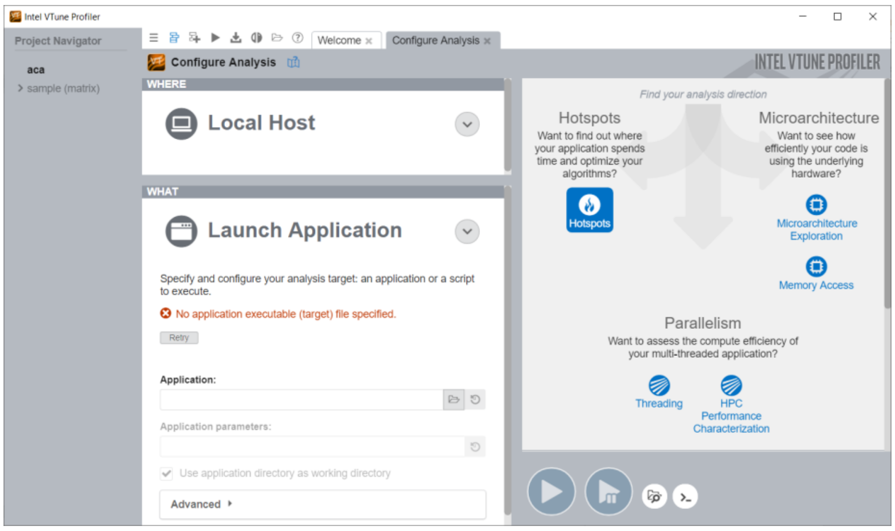


Click "**New Project**" to create a new project and store results there.

2.2.3 Configure and Run Remote Analysis

Prerequisite: For a collection to run in TCP/IP mode, ensure that Sampling Driver and Analysis Communication Agent are running on the remote system.

1. On the welcome screen, click on the **Configure Analysis** button to start a new collection.


Intel VTune Profiler
Project Navigator
aca
sample (matrix)
Welcome
Configure Analysis
Configure Analysis
INTEL VTUNE PROFILER
WHERE
Local Host
WHAT
Launch Application
Find your analysis direction
Hotspots
Want to find out where your application spends time and optimize your algorithms?
Hotspots
Microarchitecture
Want to see how efficiently your code is using the underlying hardware?
Microarchitecture Exploration
Memory Access
Parallelism
Want to assess the compute efficiency of your multi-threaded application?
Threading
HPC Performance Characterization
Retry
Advanced

Specify and configure your analysis target: an application or a script to execute.

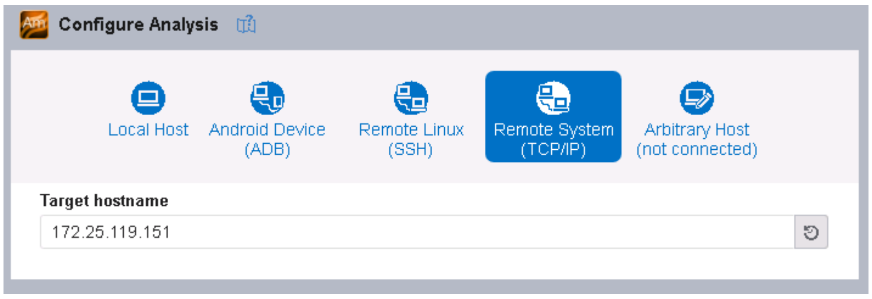
No application executable (target) file specified.

Application:

Application parameters:

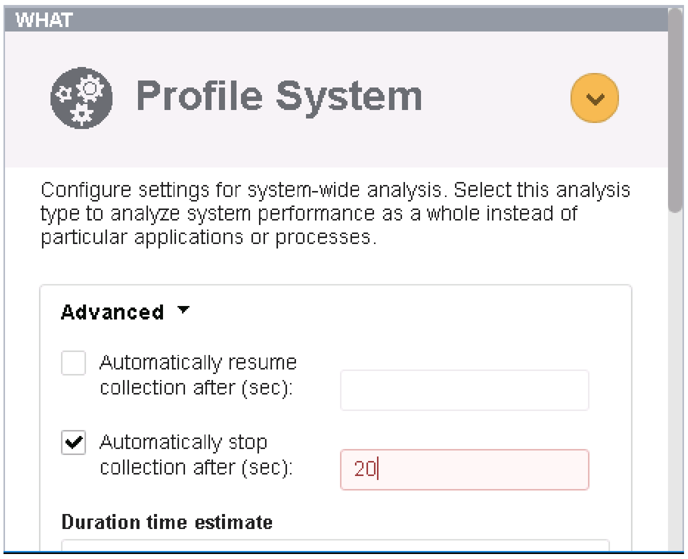
✓ Use application directory as working directory

2. In the **WHERE** pane,
   - Choose Remote System (TCP/IP).
   - Provide the IP address of the target system.



Target hostname: 172.25.119.151

3. In the **WHAT** pane, optionally set configuration to start and stop the collection.

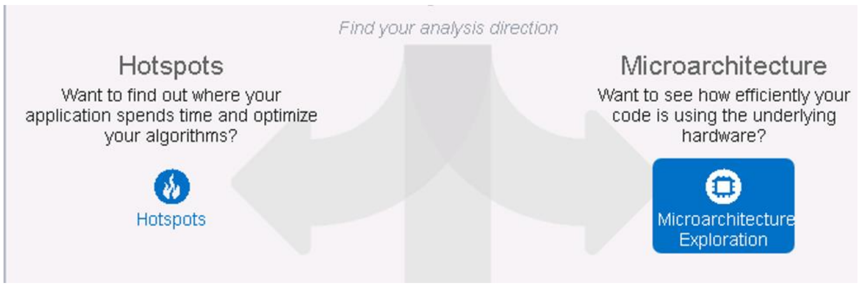


Configure settings for system-wide analysis. Select this analysis type to analyze system performance as a whole instead of particular applications or processes.

☐ Automatically resume collection after (sec):

✓ Automatically stop collection after (sec): 20

4. In **HOW** pane, select an analysis type. In TCP/IP mode, you can select between **Hotspots** and **Micro-architecture exploration**.



Find your analysis direction

Hotspots

Want to find out where your application spends time and optimize your algorithms?

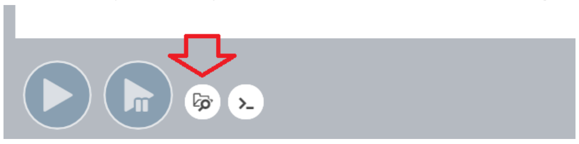
Hotspots

Microarchitecture

Want to see how efficiently your code is using the underlying hardware?

Microarchitecture Exploration

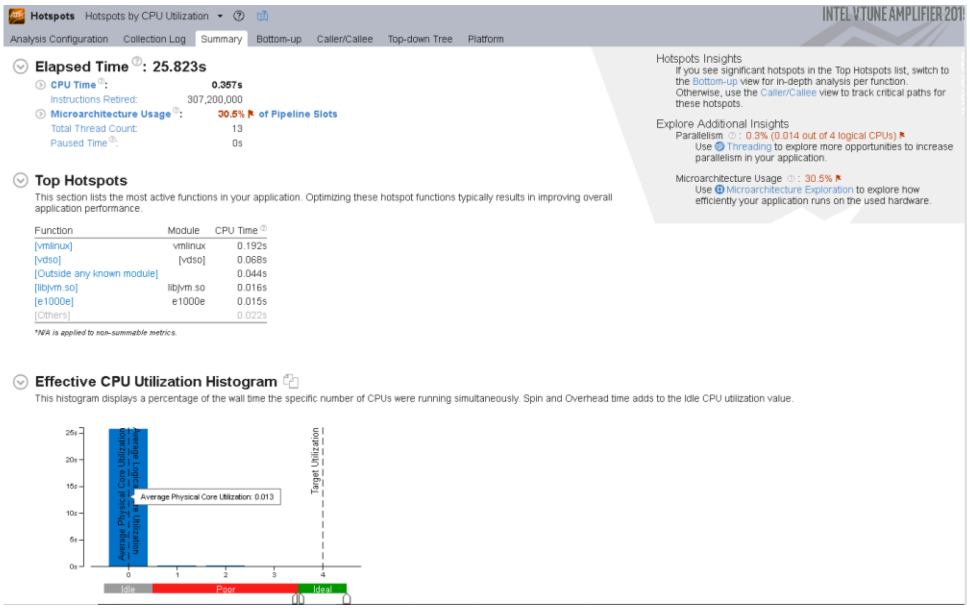
5. Configure search directories to provide paths on the host to the target binaries and symbols:



6. Click the play button to start the collection. You can stop the collection at any time by pressing the stop button. If you configured the collection to run for a specific period of time, the collection stops after that duration.

2.2.4 View Collected Data

- *Hotspots analysis*



Hotspots Hotspots by CPU Utilization

Analysis Configuration Collection Log Summary Bottom-up Caller/Callee Top-down Tree Platform

INTEL VTUNE AMPLIFIER 201

Elapsed Time: 25.823s

CPU Time:	0.357s
Instructions Retired:	307,200,000
Microarchitecture Usage:	30.5% of Pipeline Slots
Total Thread Count:	13
Paused Time:	0s

Top Hotspots

This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.

Function	Module	CPU Time
[vmlinux]	vmlinux	0.192s
[vdso]	[vdso]	0.068s
[Outside any known module]		0.044s
[libjvm.so]	libjvm.so	0.016s
[e1000e]	e1000e	0.015s
[Others]		0.022s

*N/A is applied to non-summable metrics.

Hotspots Insights

If you see significant hotspots in the Top Hotspots list, switch to the Bottom-up view for in-depth analysis per function. Otherwise, use the Caller/Callee view to track critical paths for these hotspots.

Explore Additional Insights

Parallelism: 0.3% (0.014 out of 4 logical CPUs)

Use Threading to explore more opportunities to increase parallelism in your application.

Microarchitecture Usage: 30.5%

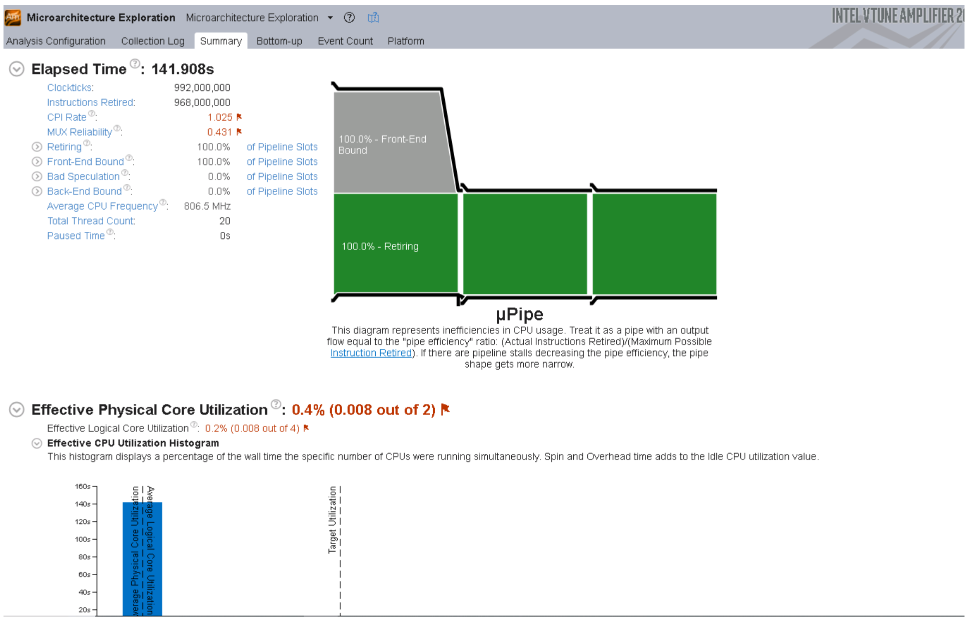
Use Microarchitecture Exploration to explore how efficiently your application runs on the used hardware.

Effective CPU Utilization Histogram

This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously. Spin and Overhead time adds to the Idle CPU utilization value.

25s
20s
15s
10s
5s
0s
0
1
2
3
4
Average Physical Core Utilization
Average Logical Core Utilization
Average Physical Core Utilization: 0.013
Target Utilization
Idle
Poor
Ideal

- *Microarchitecture exploration analysis*



Microarchitecture Exploration | Microarchitecture Exploration

Analysis Configuration | Collection Log | Summary | Bottom-up | Event Count | Platform

INTEL VTUNE AMPLIFIER 2

Elapsed Time: 141.908s

Clockticks:	992,000,000	
Instructions Retired:	968,000,000	
CPI Rate:	1.025	
MUX Reliability:	0.431	
Retiring:	100.0%	of Pipeline Slots
Front-End Bound:	100.0%	of Pipeline Slots
Bad Speculation:	0.0%	of Pipeline Slots
Back-End Bound:	0.0%	of Pipeline Slots
Average CPU Frequency:	806.5 MHz	
Total Thread Count:	20	
Paused Time:	0s	

µPipe

This diagram represents inefficiencies in CPU usage. Treat it as a pipe with an output flow equal to the "pipe efficiency" ratio: (Actual Instructions Retired)/(Maximum Possible Instruction Retired). If there are pipeline stalls decreasing the pipe efficiency, the pipe shape gets more narrow.

Effective Physical Core Utilization: 0.4% (0.008 out of 2)

Effective Logical Core Utilization: 0.2% (0.008 out of 4)

Effective CPU Utilization Histogram

This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously. Spin and Overhead time adds to the Idle CPU utilization value.

160s
140s
120s
100s
80s
60s
40s
20s
Average Physical Core Utilization
Average Logical Core Utilization
Target Utilization

3 Architecture

3.1 Hardware PMU Counters

Intel hardware contains Performance Monitoring Units (PMUs) that facilitate the collection of performance data when the processor executes instructions. Each PMU has multiple hardware counters that collect performance data under various execution scenarios. Each scenario is called an event. For example, L3 Cache Miss is an event.

The hardware counters can be:

- Fixed function (FF) counters
- General purpose (GP) counters

A fixed function counter counts three specific events – retired instructions, reference clocks, and core clocks. However, you can program a general-purpose counter to count several hundreds of events.

For more information on PMUs and supported platforms, see chapter 18 of the Intel® 64 and IA-32 Architectures Software Developer's Manual – Volume 3.

3.2 Sampling Target Collector Design

3.2.1 Overview

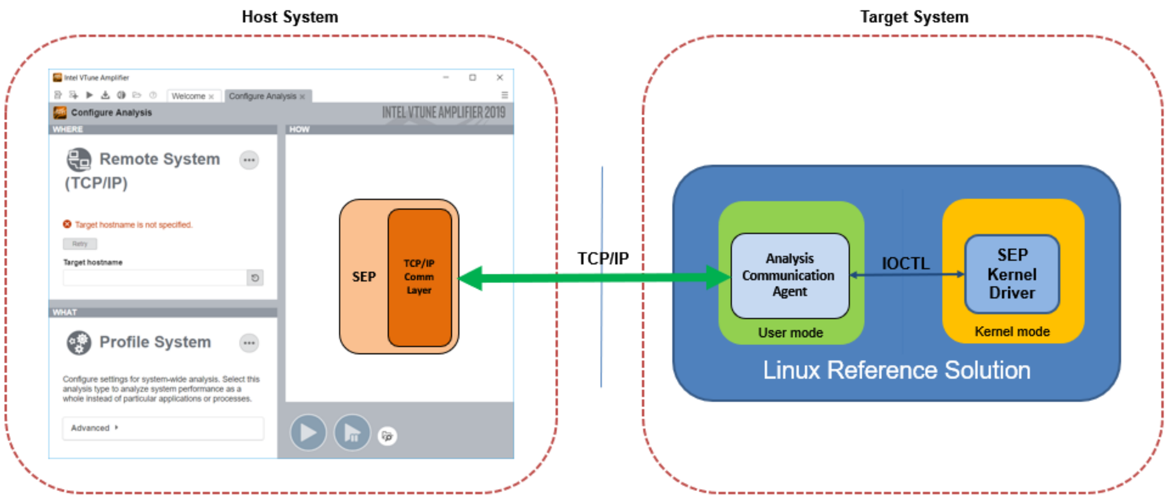
Intel VTune Profiler has a collector called Sampling Collector that leverages hardware PMU counters to collect performance data. In this workflow,

1. The counters are programmed to overflow on reaching a fixed number of events.
2. When the counters overflow based on the event condition, a performance monitoring interrupt is generated.
3. During the interrupt, a sample is recorded that includes the execution context and other performance data.

4. The counters are reset.

The workflow repeats until the collection ends. It also collects the process/module information of the processes that are running during the collection. This information is processed offline with sampling data to generate performance analysis. The sampling collection depends on various mechanisms to collect the required data.

3.2.2 Analysis Communication Agent Reference



Target hostname is not specified.

Configure settings for system-wide analysis. Select this analysis type to analyze system performance as a whole instead of particular applications or processes.

Fig 2. Target Sampling Collector

On the target, the Sampling Collector has:

- Analysis Communication Agent running in the user space.
- A Sampling Driver running in the kernel space.

NOTE: The reference implementation is based on Linux OS and requires two components - Analysis Communication Agent running in the user-mode and Sampling Driver running in the kernel-mode, but this structure depends on the target OS. In some RTOS platforms, these two components may have to be combined into one.

You can use Intel VTune Profiler on the host to start collections on the remote target. When this happens, the SEP collector (operating inside Intel VTune Profiler on the host) can build the PMU programming based on PMU events specified for the hardware platform. The sampling driver in the kernel space (target) then assigns the required registers to program the PMU. The SEP collector program in the user space (host) gets streams of data from the kernel space (target). It combines the data into a single binary data file that you can see in Intel VTune Profiler.

Analysis Communication Agent starts a TCP/IP-based socket connection server at a specific port. Once running, it listens for any connections from a Sampling collector running on a host system. An instance of Analysis Communication Agent can serve multiple connections but only one connection can be active at any time.

4 Porting Requirements

4.1 Hardware Access Mechanisms

You need several low level access mechanisms to hardware in order to capture appropriate profiling data.

- **CPUID Information**
  Sampling Driver needs to gather several types of data to determine hardware configurations and programming.
- **PMU (Performance Monitoring Unit)**
  Many types of PMU registers need to be programmed. The PMU resources allow to collect core and uncore performance event data via different access mechanisms:
  - Read/Write MSR (Model Specific Register) to access Core PMU and some Uncore PMUs
  - Read/Write to PCI (Peripheral Component Interconnect) to some Uncore PMUs
  - Memory Mapped I/O to access some Uncore PMUs
- **Advanced Programmable Interrupt Controller (APIC)**
  The Sampling Driver should be able to access APIC registers.
  - To obtain CPU topology information, you must be able to read the APIC_ID.
  - You can access xAPIC (Extended APIC) through Memory Mapped I/O.
  - You can access x2APIC via MSRs (if supported).

  For more details, see the Intel® Architecture Software Developer Manual.

- **Performance Monitoring Interrupt (PMI)**
  When profiling core events, the PMI hardware interrupt is used to collect sample data. If the non-maskable interrupt (NMI) mode is supported on the target OS, it is preferable (although not mandatory) that you enable NMI mode. In NMI, the profiled data will provide visibility into the interrupt masked regions also. Linux provides the kernel API to register a kernel callback function based on NMI. The callback function will be invoked by the OS whenever an NMI is generated. Use APIC programming to implement the PMI in NMI mode. See the Intel® Architecture Software Developer Manual for more information.

4.2 Operating System Services

Along with hardware data, you must capture software context information. Some OS services are also required to program data collection in hardware.

- **Software Context Information**
  Software context information includes the identifier and the memory information of the software managed by the OS during runtime. Linux OS loads and unloads software as module. It manages these modules as process and thread at runtime. Therefore, the reference implementation collects all modules, processes, and threads that run when profiling hardware data.
  - **Module Information** – Register with the OS to get process/module information of the processes that were launched dynamically. The OS should provide a hook to register a callback function. The callback function in the kernel space is then invoked any time a new process is launched. The callback notification should include the process information described in Section 6.2. Inside the callback function, the process/module information should be recorded into the task information buffer setup before starting the collection.
  - **Process & Thread Enumeration** – The OS should provide the list of processes running along with the information of each process. This happens once - at the start or end of the collection. The OS records the information into the process information buffer.
- **Parallel function calls across multiple CPUs**
  If the target supports Symmetric Multiprocessing (SMP) on more than one CPU, it is required to perform certain tasks across all CPUs simultaneously. For example, it is essential to read the Timestamp Counter (TSC) across all CPUs during data collection. The reference solution leverages Linux kernel API to satisfy this requirement.

- **Contiguous memory allocation**
  Some advanced hardware facilities such as Precise Event Based Sampling (PEBS) require that memory buffer allocation must be mapped to contiguous linear addresses. See the Intel® Architecture Software Developer Manual for more information.

5 Collection Workflow

Figure 3 describes the sampling collection workflow that involves:

- Intel VTune Profiler
- SEP Collector
- Analysis Communication Agent
- Sampling Driver.

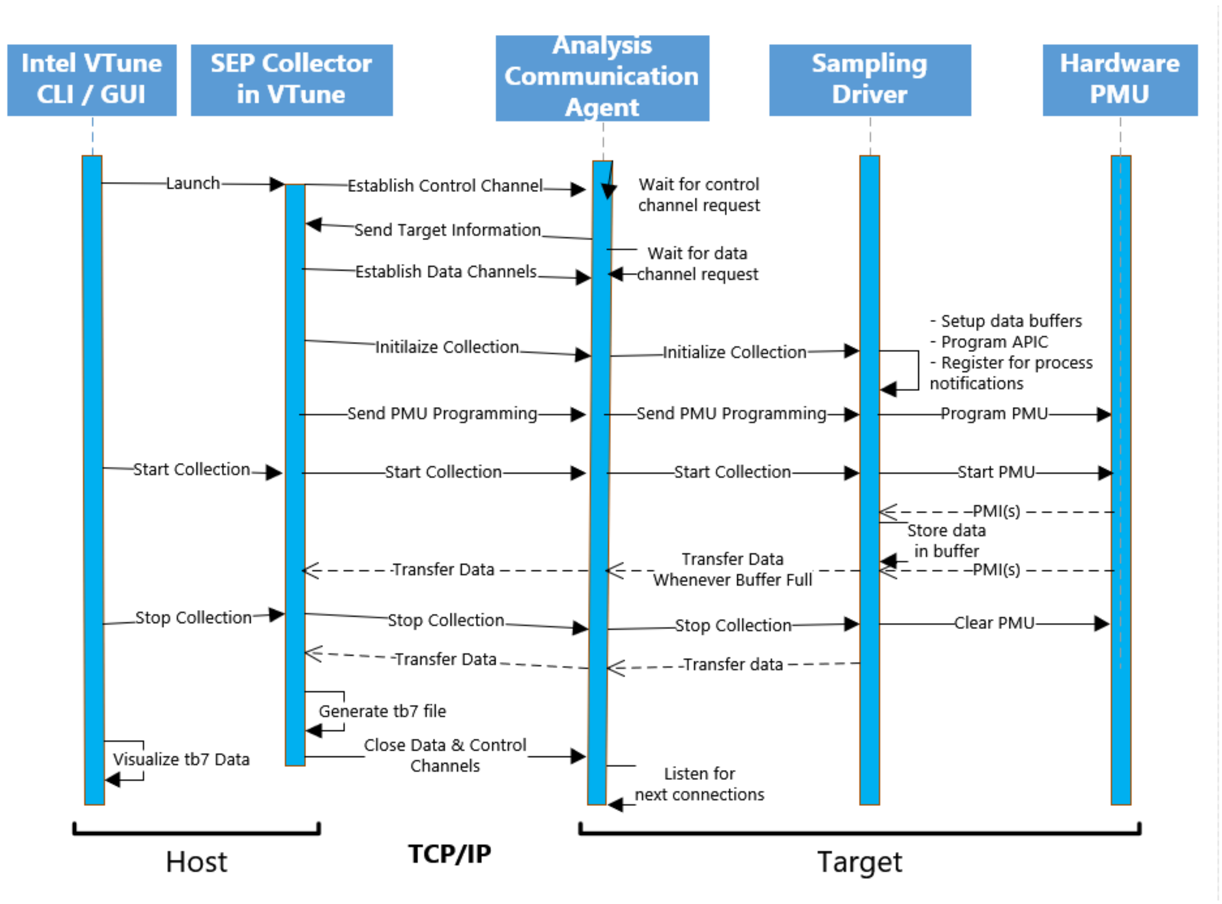


Fig. 3. Sampling Collection Workflow

1. The SEP collector communicates with Analysis Communication Agent using the communication library in the profiling agent package.
2. The communication library sends and receives control and data messages between host and target using TCP/IP protocol.
   - If the target OS supports the POSIX socket API, there is no need to port the communication library. The library can be recompiled as is with your target application build toolchain.

    - If the target OS does not support the POSIX socket API, you must port the communication library using the relevant API that supports TCP/IP socket programming.
3. The communication library allows the SEP collector (on host) to deliver commands to Analysis Communication Agent (on target) to control profiling behaviors.
4. The communication library also provides the interface to receive data from Analysis Communication Agent.
5. Multiple data channels are established with one channel for sending control commands and several data channels to transfer the data collected.
    - Different data channels are dedicated to handle single data type from the designated data source.
    - If more than one CPU is available, each CPU has its own data channel.
    - Two additional channels are enabled to transfer non-core and module data.
    - The number of channels to be established is dynamically determined based on the number of data sources available.
    - The data channels support two data transfer - immediate transfer and delayed transfer. In the immediate transfer mode, the host receives data from the target as soon as it is generated. In the delayed transfer mode, data is stored in temporary files during data collection. The host receives the data right after the collection completes.

6 Required Data for Performance Profile

This section describes the types of data that need to be collected to prepare a performance profile.

6.1 Platform Topology

The platform topology helps with interpreting the collected data. The topology is queried using the CPUID instruction. The topology information contains:

- Number of packages
- Number of CPUs
- Number of logical cores
- Support for hyperthreading

6.2 Process, Thread and Module Information

For dynamically loaded processes, the OS should provide these process/module details in the callback function.

- Process/Thread ID
- Process/Thread Name
- Executable Name – executable binary corresponding to the process name
- Module Segment start address – start of module segment address range within the process virtual memory
- Module Segment end address – end of module segment address range within the process virtual memory
- Time Stamp when the process/thread starts (can be 0 if dynamic process/thread creation is not supported)
- Time Stamp when the process/thread stops (can be 0xFFFFFFFFFFFFFFFF if dynamic process/thread creation is not supported)
- Time Stamp when the module is loaded into process virtual address space (can be 0 if dynamic module loading is not supported)
- Time Stamp when the module is unloaded from the process virtual address space (can be 0xFFFFFFFFFFFFFFFF if dynamic module loading is not supported)

All this information is necessary to correctly map sample instruction pointer to the actual instruction in the binary file and provide hotspots on process/thread/module/function/source levels.

6.3 Performance Data

Performance data is saved into the sampling buffers inside the performance monitoring interrupt (PMI). The data includes:

- Event ID – the event that caused the counter overflow
- Instruction Pointer
- CPU Number
- Process ID
- Thread ID
- Time Stamp
- 32-bit or 64-bit task
- RFLAGS information
- PEBS Information (advanced mode)
- Last Branch Records data (advanced mode)

7 Advanced Features

7.1 Precise Event Based Sampling (PEBS)

When an instruction causes an event, the event counter overflows. When this occurs, the PEBS feature generates several records that capture architectural state information. This information is more accurate than the data captured inside PMI due to a delay in interrupt delivery. The PEBS feature can collect data without using NMI and even when interrupts are masked. Therefore, it is preferable (although not mandatory) to enable collection through PEBS, if your OS can reserve contiguous memory.

For the PEBS configuration, records are stored in a memory buffer in kernel space. The configuration requires a setup of contiguous memory space. The memory buffer should be registered with PEBS registers. The hardware saves PEBS data into the registered memory buffer. This data should be read during the performance interrupt and transferred into the sampling data buffer.

PEBS is a non-architectural feature in Intel processors. Therefore, the implementation can vary by architecture. You must enable PEBS data collection to perform Microarchitecture Exploration and Memory Access analyses using Intel VTune Profiler.

For more information on PEBS, see Intel® Architecture Software Developer Manual, Volume 3.

7.2 Last Branch Record (LBR)

Intel hardware provides the ability to record the last taken branches, interrupts, and exceptions to a stack of hardware msr registers. The LBR tracks branch instructions (like JMP, Jcc, LOOP and CALL instructions) and other operations that cause a change in the instruction pointer (external interrupts, traps, and faults). The size and exact locations of the LBR stack are generally specific to a hardware platform family and model.

You must enable the LBR feature to collect data through Intel VTune hardware callstacks.

7.3 Event Multiplexing

Modern Intel CPUs have four PMU counters per core that allow only four events to be collected simultaneously. However, there can be over 100 events available at any given instant in time. Frequently, you may need to collect data from multiple events in a single profiling. To work around the hardware limitation, Intel VTune Profiler has an event multiplexing feature when the number of events to profile exceeds the number of available hardware counters.

The events are split into multiple groups and the groups are scheduled for data collection in a round-robin manner. Based on the configuration, you can switch the groups every few milliseconds. At the end of the collection, there is data from all of the events in all of the groups.

You must enable event multiplexing through the command line to perform microarchitecture analysis with Intel VTune Profiler.

7.4 Uncore Profiling

The uncore consists of the hardware units beyond the cpu cores. The performance units in the uncore facilitate the system-wide performance visibility.

The number of available uncore PMUs and their access mechanism (MSR, PCI, or MMIO) are hardware platform model specific. For more information, see documentation on uncore PMUs.

You must enable support for uncore PMUs to perform Microarchitecture Exploration analysis.

8 Source directory structure

8.1 Top level source structure

The top-level source structure mainly consists of agentdk and sepdk folders.

```
├── agentdk
├── docs
├── sepdk
├── tests
├── CONTRIBUTING.md
├── LICENSE.txt
└── README.md
```

8.2 Source structure for agendk

The agentdk is the user-mode part of the source. This provides the functionality to communicate with host system and also with the sampling kernel driver. It establishes the communication channels between host system and target system to control the data collection and to transfer the data.

```
agentdk
├── abstract.c
├── abstract.h
├── abstract_service.c
├── abstract_service.h
├── collection_traces.c
├── collection_traces.h
├── communication.c
├── communication.h
├── include
│   ├── lwpmudrv_defines.h
│   ├── lwpmudrv_ecb.h
│   ├── lwpmudrv_ioctl.h
│   ├── lwpmudrv_struct.h
│   ├── lwpmudrv_types.h
│   ├── lwpmudrv_version.h
│   └── rise_errors.h
├── log.h
├── Makefile
├── sepagent.c
├── sepagent_parser.c
└── sepagent_parser.h
```

8.3 Source structure for sepdk/include folder

The sepdk folder contains source for the sampling driver kernel component. The header files shared between and user-mode and driver are located in sepdk/include folder.

```
sepdk/include
├── error_reporting_utils.h
├── lwpmudrv_defines.h
├── lwpmudrv_ecb.h
├── lwpmudrv_gfx.h
├── lwpmudrv_ioctl.h
├── lwpmudrv_struct.h
├── lwpmudrv_types.h
├── lwpmudrv_version.h
├── pax_shared.h
└── rise_errors.h
```

8.4 Source structure for sepdk/src folder

The source code for the sampling driver implementation is in the sepdk/src folder. The implementation details of the sampling driver are described in the following sections.

```
sepdk/src
├── apic.c
├── boot-script
├── build-driver
├── control.c
├── core2.c
├── cpumon.c
├── eventmux.c
├── inc
├── insmod-sep
├── linuxos.c
├── lwpmudrv.c
├── Makefile
├── output.c
├── pci.c
├── pebs.c
├── perfver4.c
├── pmi.c
├── pmu_list.c
├── README.txt
├── rmmod-sep
├── sepdrv_p_state.c
├── silvermont.c
├── sys32.S
├── sys64.S
├── sys_info.c
├── unc_common.c
├── unc_gt.c
├── unc_mmio.c
├── unc_msr.c
├── unc_pci.c
├── unc_power.c
├── unc_sa.c
├── utility.c
└── valleyview_sochap.c
```

9 Implementation Details

The reference implementation contains two components:

- Analysis Communication Agent program runs in the user mode.
- Sampling Driver runs in the kernel mode.

The sections in this topic describe the interfaces required for each component.

Depending on the architecture of the operating system, you can also consider implementing the two components as a single program.

9.1 Analysis Communication Agent

9.1.1 Overview

Analysis Communication Agent is a program that runs on the target system that performs these functions:

- Listen for connections from host (socket-based connections).
- Establish control channel.

- Establish multiple data channels to transfer data.
- Command a pass-through from host to sampling kernel driver on the target and vice-versa.
- Transfer the data collected in Sampling Driver to the host.

9.1.2 Communication Protocol

The communication between host and target is designed to work on TCP/IP protocol. Therefore, you can establish communication if the target OS supports socket-based communication using TCP/IP protocol. Analysis Communication Agent uses an implementation based on the POSIX API. If the target OS supports the POSIX API, you can enable communication without modification. If there is no support for POSIX API, you must port the communication library with its own socket API.

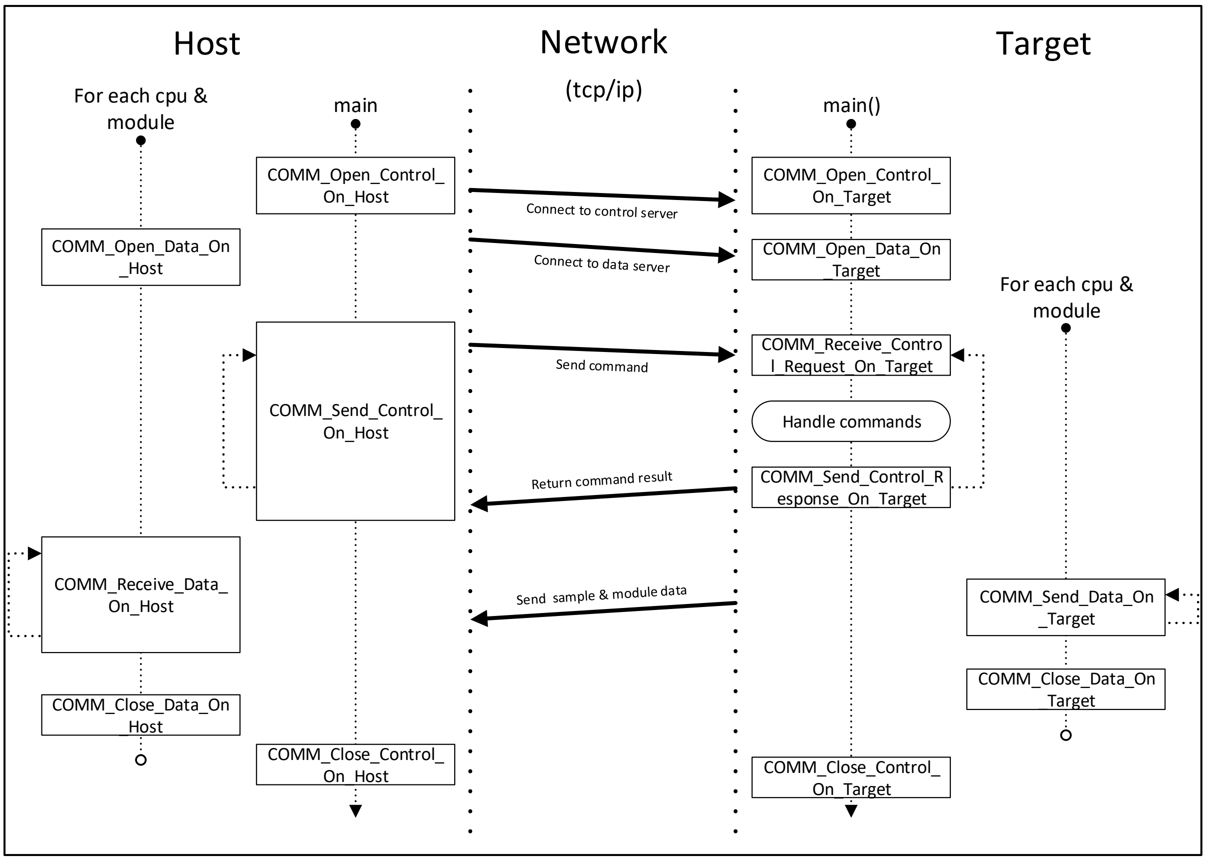

Host
Network
Target
(tcp/ip)
For each cpu & module
main
main()
COMM_Open_Control_On_Host
COMM_Open_Control_On_Target
Connect to control server
COMM_Open_Data_On_Host
COMM_Open_Data_On_Target
Connect to data server
For each cpu & module
Send command
COMM_Receive_Control_Request_On_Target
COMM_Send_Control_On_Host
Handle commands
Return command result
COMM_Send_Control_Response_On_Target
COMM_Receive_Data_On_Host
Send sample & module data
COMM_Send_Data_On_Target
COMM_Close_Data_On_Target
COMM_Close_Data_On_Host
COMM_Close_Control_On_Host
COMM_Close_Control_On_Target

Fig 4: Communication flow of Analysis Communication Agent using POSIX API

As described in Figure 4, to establish an initial connection between host and target, the target opens TCP/IP sockets and receiving connection requests from the host. This prevents a firewall issue on host side where incoming connections on host are blocked. To set up the initial connection, the host requires the target IP address and port number.

In Analysis Communication Agent, there are two files “communication.h” and “communication.c” for the communication API. If you need to port it to socket API supported by the OS, replace the following POSIX API in these files:

socket	Create an endpoint for communication and returns a descriptor.
setsockopt	Set options on sockets
bind	Bind a name to a socket
listen	Listen for connections on a socket
accept	Accept a connection on a socket
connect	Initiate a connection on a socket

send	Send a message on a socket
recv	Receive a message from a socket
close	Close a socket descriptor

9.1.3 Communication Channels

There are two types of communication channels that are established between host and target:

- **Control Channel**: This channel is used to send control commands from SEP (on the host) to Analysis Communication Agent (on the target) which is forwarded to Sampling Driver. Some control commands may need to send the command output back to the host. Only one control channel exists during collection.
- **Data Channel**: This channel is used to transfer data from target to host. For a single collection, multiple data channels can be established, each dedicated to CPU core, uncore, and module data.

9.1.4 Porting Requirements

9.1.4.1 System Information

Information like cupid or reference frequency are captured using CPUID instructions. Alternatively, you can use OS supported functions to get the same information.

These instructions are implemented in the *sepagent.c file*. These functions are implemented as inline assembly functions.

sepagent_Read_Cpuid	Uses CPUID instruction from x86 architecture to discover cpuid of the processor
sepagent_Get_Tsc_Frequency	Uses CPUID instruction from x86 architecture to discover processor reference frequency

9.1.4.2 Communication API's

Use communication APIs to:

- Establish a TCP/IP connection
- Create communication channels
- Communicate with SEP
- Transfer data
- Close a TCP/IP connection

Communication APIs are typically implemented in the *communication.c* file. Socket APIs are used internally by all the functions. To use Analysis Communication Agent, your operating system must support socket APIs.

This table describes common communication APIs described in *communication.c*.

COMM_Open_Control_On_Target	Creates a socket, binds and waits for connections request from host. Upon connection request from SEP/host it accepts connections and creates control communication channels. There is an initial exchange of information between SEP and Analysis Communication Agent here.

COMM_Receive_Control_Request_On_Target	Receives control commands and any additional buffer on control channel and processes it to send to sampling driver
COMM_Send_Control_Response_On_Target	Sends the response to control command received along with command status and sometimes additional buffer
COMM_Close_Control_On_Target	Closes the control communication channel
COMM_Open_Data_On_Target	Establishes data channels for specified data type and identifies it using connection id.
COMM_Send_Data_On_Target	Sends the data to the SEP/host over specified connection id
COMM_Close_Data_On_Target	Closes the data communication channel

9.1.4.3 IOCTL Handling

IOCTL is a system call for input and output operations. If OS has a separation between user and kernel spaces, IOCTL is used to communicate between Analysis Communication Agent (user space) and Sampling Driver (kernel space).

If the target OS does not have this separation, the user and kernel spaces can be connected through function calls.

All IOCTLs are defined in the *include/lwpmudrv_ioctl.h* header file.

Most IOCTL handling functions are implemented in *abstract_service.c*. This file has functions to:

- Open the device driver
- Communicate IOCTL
- Close device driver

This table describes commonly used IOCTL functions.

abstract_Open_Device_Driver	Returns handle to the specified device-driver, which can subsequently be used for calls to “Device Input and Output Control” (IOCTL)
ABSTRACT_Send_IOCTL	Performs the IOCTL specified by the control code and returns. Expects to pass an in buffer and get data back through the out buffer.
ABSTRACT_Open_Driver	This function opens the driver to perform work. Internally calls abstract_Open_Device_Driver()
ABSTRACT_Close_Driver	This function closes the driver for the current invocation. The driver itself will not be unloaded.

9.1.4.4 Data transfer to SEP

If the OS has a separation between user and kernel spaces, the data generated by Sampling Driver during collection must be copied from kernel space to user space and eventually to SEP over the communication layer. If there is no separation between the two spaces, the collected data can be sent to SEP directly.

Data transfer happens through these modes:

Immediate transfer mode - Read data from kernel space continuously. Send read data to SEP simultaneously.

Delayed transfer mode - Read data from kernel space continuously and store it in temporary files. At the end of collection, the data stored in temporary files is sent to SEP. This requires minimum space availability on the target.

The functions listed here are implemented in the *abstract.c* file.

abstract_Send_Data_To_Host	Sends data form specified file to remote host
abstract_Read_Records	Reader thread for each device / temp file that needs to be created
abstract_Spawn_Pthreads	Spawn threads to create the sample and module temp files. Allocate and set up per-thread argument structures and pthread data structures, later used for joining in abstract_Join_Pthreads()
abstract_Spawn_Pthreads_UNC	Spawn threads to create the uncore sample temp files. Allocate and set up per-thread argument structures and pthread data structures, later used for joining in abstract_Join_Pthreads_UNC()
abstract_Join_Pthreads	Wait for the threads spawned in abstract_Spawn_Pthreads(). Free per-thread argument and pthread structures
abstract_Join_Pthreads_UNC	Wait for the threads spawned in abstract_Spawn_Pthreads_UNC(). Free per-thread argument and pthread structures
abstract_Start_Threads	All buffers and temp files are set up and the system will be prepared to start collection
abstract_Start_Threads_UNC	All buffers and temp files for uncore are set up and the system will be prepared to start sampling.
abstract_Stop_Threads	Stop all the threads.

9.1.4.5 Version Information

Version information includes:

- Version of Sampling Driver
- Information about processors that are online and configured
- TSC frequency
- Driver setup information
- Virtualization availability
- Vendor information

You can find implementation details for version information in *sepagent.c* and *abstract.c*.

sepagent_Print_Version	Prints the version information when –v option is provided
ABSTRACT_Num_CPUs	Get the number of cores in system
ABSTRACT_Version	Get the version number of the kernel mode driver
ABSTRACT_Get_Drv_Setup_Info	Get driver setup information, this populates DRV_SETUP_INFO_NODE data structure

9.2 Sampling Driver

9.2.1 Overview

Sampling Driver provides the functionality to program the hardware PMU registers and collect performance data. Some parts of the Sampling Driver source code are OS-specific while other parts are OOS-independent. The implementation of OS-specific components is specific to the RTOS architecture and can vary depending on the contents of the target OS.

9.2.1.1 Implementation of OS-specific Components

File Name	Description
apic.c	Programming APIC
cpumon.c	PMU or NMI handler registration
linuxos.c	Module/Process data collection implementation
control.c	Common API for memory allocation/deallocation
pmi.c	Performance Monitoring Interrupt Handler
pci.c	API for accessing PCI configuration space and Memory Mapped I/O
sys_info.c	System topology information
sys32.S	Assembly code for accessing MSR
sys64.S	Assembly code for accessing MSR
utility.c	API for handling specific dispatch tables for accessing various PMU programming
pebs.c	PEBS implementation

9.2.1.2 Implementation of OS-independent Components

File Name	Description
lwpmudrv.c	IOCTL handler, collection control flow, and PMU programming
core2.c	
perfver4.c	Implementation for collection control flow for Perfmon Version 4 based platforms
output.c	Buffer Handling for the collected data
eventmux.c	Event multiplexing handler
unc_common.c	Common functionality to support uncore performance data
unc_msr.c	Uncore performance data collection for MSR based PMU registers
unc_pci.c	Uncore performance data collection for PCI based PMU registers
unc_mmio.c	Uncore performance data collection for MMIO based PMU registers
unc_power.c	Support for collecting power related registers

Enabling support for each of the analysis types described in Section 1.4 requires specific implementation in the Sampling Driver.

The following sections describe the implementation details organized by the Intel VTune Profiler analysis profile types.

9.2.2 Enable Hotspot Analysis

9.2.2.1 System Information

Hardware information is captured through the CPUID instruction. This instruction returns processor identification and feature information based on the input value entered in the EAX register. The captured information can include details like:

- CPU model
- CPU topology
- Core PMU information

Sampling Driver captures all of the information from different input values available on the hardware. If the hardware has multiple CPUs, Sampling Driver captures it across all CPUs. Therefore, you need a parallel function call across all CPUs to obtain the information as described in the porting requirements (Section 9.1.4) above.

In the driver source, the *sys_info.c* file captures CPUID information. To execute CPUID instruction, this file relies on the *UTILITY_Read_Cpuid* macro function that is defined in *utility.c*. If the target OS provides an equivalent API to directly query the CPUID information, use the API instead.

9.2.2.2 APIC programming

You need APIC programming to enable the PMI and register the interrupt handler. Sampling Driver also captures the APIC identifier for each CPU to construct CPU topology. On Intel hardware, you can access the APIC through MMIO or MSR access mechanisms. The choice of access mechanism depends on CPUID information, which in turn is based on the availability of x2APIC. For more information, see Intel® Architecture Software Developer Manual (Volume 2).

In the driver source, the *apic.c* file handles APIC programming. This Linux target reference solution relies on a Linux kernel API to obtain the APIC identifier and enable/disable PMI. Replace with the target OS API as it applies to your case.

9.2.2.3 Read/Write MSR

MSR read/write operations are essential services for profiling. Typically, the OS provides the API for the reading and writing of MSR registers. This Linux target reference solution uses the kernel API for MSR accesses such as `rdmsr()` or `wrmsr()`. There are wrapper functions called `SYS_Read_MSR` and `SYS_Write_MSR` that are defined in `utility.c`. Use OS-specific functions inside these wrapper functions.

9.2.2.4 Handle IOCTL Calls

IOCTL is a system call for input and output operations. If the target OS has a separation between user and kernel spaces, IOCTL is used to communicate between Analysis Communication Agent in the user space and Sampling Driver in the kernel space. If there is no separation, the communication happens through function calls.

In this Linux target reference solution, all IOCTL calls are defined in `include/lwpmudrv_ioctl.h`. They are handled in the file `lwpmudrv.c.` Search for `lwpmu_Service_IOCTL` for more information.

9.2.2.5 Program Core PMU

The Programming Core PMU is implemented in `silvermont.c`, `core2.c`, and `perfver4.c`. Depending on the CPU architecture that CPUID detects, Sampling Driver uses one of these files to enable the Programming Core PMU. These files are OS-agnostic, so the codes do not need to get ported to a different OS.

9.2.2.6 Interrupt Handler

When you port to a target OS, enabling the PMI is the most critical aspect of this operation. This is because the registration and handling of interrupts varies substantially by OS. In this Linux reference, Sampling Driver implements the interrupt handler in `pmi.c`. The handler registration is implemented in `cpumon.c`.

There are two types of interrupt modes:

- Maskable interrupts
- Non-maskable interrupts (NMI)

Since Linux OS supports both modes, the Linux target reference solution uses NMI as the PMI delivery method. The advantage of using NMI is the profiling can capture data samples although interrupts are masked by other software execution. Otherwise, when profiled, the data sample can point to an incorrect software location. Interrupt priority is also important in this regard. If the PMI is set to a low priority, the data samples will not be appropriately recorded.

In the interrupt handler, several types of information are captured and stored as a sample in the memory buffer. The sample includes:

- Timestamp Counter (TSC) value
- CPU ID
- CS and EFLAGS registers
- Instruction pointer (IP) address
- Process and thread ID – These are used to associate with software codes when module information is captured separately.

9.2.2.7 Module information

Module information is used to pinpoint the software codes that get executed where the PMI takes a PMU sample. The module data and the PMU sample are stored separately during collection. During post-processing, they are associated based on the process and thread ID.

Module information includes:

- Process ID
- TSC values where the module is loaded/unloaded
- Memory base address
- Length in memory,

The target OS may have a different terminology or concept of process handling. However, you must map the target OS to the Linux OS process, thread, and module structure. If there is no thread support, the thread ID can always be 0, process ID can also be always 0 if there is just a single executable address space.

In this Linux target reference solution, Sampling Driver implements the module data collection in the `linuxos.c` file. Search for `MODULE_RECORD_` to populate the correct information for your target OS.

9.2.3 Enable Microarchitecture Exploration analysis

9.2.3.1 Precise Event Based Sampling (PEBS)

Support for PEBS support is implemented in the `pebs.c` file in Sampling Driver. Most of these functions are independent of the OS. However, two functions need to be ported to the target OS. They are:

- PEBS_Allocate
- PEBS_Deallocate

PEBS requires continuous memory allocation to set up the DS save area. If the target OS supports the memory allocation and deallocation required by PEBS, they can be ported to your target OS. For information on requirements for programming PEBS, see **Setting Up the DS Save Area** in the Intel® Architecture Software Developer Manual.

9.2.3.2 Event Multiplexing

Event multiplexing is independent of the operating system. The switching of the event group occurs inside the interrupt handler. In this reference solution, the interrupt handler in `pmi.c` calls the `swap_group` function which switches from one event group to another. This function is implemented in `silvermont.c, core2.c,` and `perfver4.c` files specific to hardware platform family model.

9.2.4 Enable Memory Access Analysis

The Memory Access analysis collects the memory bandwidth data. This is the number of read/write bytes accessed per second. The collection of the memory bandwidth requires read/write access mechanism for PCI and MMIO registers. The API for PCI access helps enable this analysis.

9.2.4.1 PCI Access Mechanism

The PCI access mechanism involves reading from or writing to the registers in the PCI configuration space at various offsets. A PCI device has a unique bus/device/function combination that can be used to access the configuration space. The access mechanism could be specific to an operating system.

In the case of Linux OS, the access mechanism invokes the `pci_find_bus()` system call to get the `pci_bus` structure for a specific bus. The mechanism then uses read() or write() API calls and passes a specific device, function, and offset combination. The implementation is defined in the `pci.c` file.

Typically, a base address for the list of registers is saved at an offset in the PCI configuration space. To access the `perfmon` registers exposed as a memory map:

1. Read the base address at a certain bus/device/function/offset.
2. Map the memory at this base address.

API	**Description**
PCI_Initialize	Scans the system to find the list of available buses
PCI_Read_U32	Reads a 4 byte value from a specific bus/device/function/offset
PCI_Read_U64	Reads a 8 bytes value from the specific bus/device/function/offset
PCI_Write_U32	Writes a 4 byte value to a specific bus/device/function/offset
PCI_Write_U64	Writes a 8 byte value to a specific bus/device/function/offset

9.2.4.2 MMIO Access Mechanism

In the Memory Mapped IO access mechanism, a certain length of memory is mapped, starting at a base address. The mechanism to find the base address is specific to hardware platform. Once the memory is mapped, you can access all available registers in the memory map by adding an offset at which the register is available. The implementation for mapping memory is specific to the target operating system. On Linux OS, memory mapping happens through the `ioremap_nochache()` system call. Once mapped, you can use the readl()/witel() system call to read from or write to the memory region. The following table lists the functions that need to be implemented to access the memory mapped address space.

API	**Description**

PCI_Map_Memory	Map the memory at a specific physical address to a virtual address. Used ioremap_nocache() system to perform this on Linux
PCI_MMIO_Read_U32	Reads a 4-byte value from a specific memory location
PCI_MMIO_Read_U64	Reads an 8-byte value from a specific memory location
PCI_MMIO_Write_U32	Writes a 4-byte value to a specific memory location
PCI_MMIO_Write_U64	Writes an 8-byte value to a specific memory location
PCI_Read_From_Memory_Address	Map a memory location, read a 4-byte value and unmap the memory
PCI_Write_To_Memory_Address	Map a memory location, write a 4-byte value and unmap the memory
PCI_Unmap_Memory	Unmap the virtual memory previously mapped. Uses iounmap() system call on Linux

9.2.4.3 Uncore Collection

The uncore consists of the hardware units beyond the processor cores. It is necessary to collect performance data from uncore units to understand system-wide performance behavior. This requires writing to and reading from the uncore performance hardware registers. Depending on the hardware platform, you can access these registers through MSR, PCI, or MMIO mechanisms. The implementation for the uncore collection is distributed between `unc_msr.c`, `unc_pci.c`, and `unc_mmio.c` files where each file contains the corresponding access mechanism.

The implementation involves the following steps:

1. Program the registers to configure the collection.
2. Start the collection.
3. Stop the collection.
4. Read data from the uncore registers.

The `unc_common.c` file provides the API to obtain platform topology. The workflow involves these steps:

1. Scan the PCI devices.
2. Find the devices related to uncore PMUs.
3. Identify the correct bus numbers for those devices.

This table lists the example API for PCI devices. The descriptions for the MSR and MMIO APIs are similar.

API	**Description**
unc_pci_Write_PMU	Configure the uncore PMU for the collection of specific events
unc_pci_Enable_PMU	Start the uncore collection
unc_pci_Disable_PMU	Stop the uncore collection
unc_pci_Trigger_Read	Read PMU data
unc_pci_Read_PMU_Data	Read PMU data

10 Test Utilities

Analysis Communication Agent contains basic tests to validate the behavior of the target agent. Tests emulate specific host functionalities to verify the compliance of the target agent with communication protocol and collection flow functionality. The testing infrastructure is a set of Python scripts that should be executed on the host system. To run these tests, you must have a version of Python that is 2.7 or higher.

10.1 Preparation of Target Platform

To verify the validity of collected data, you must build and execute a dedicated test application on the target before running tests.

10.1.1 Build Test Application on Target

The actual steps to compile the test application depend on the build specifics of the target system. Execute the following commands either:

- On the target directly or
- On the host under the target build environment

```
$ cd ./apps/target

$ make
```

<u>Note:</u> The test application uses the POSIX thread API to launch multiple threads. If POSIX API is not available on the target, change the test application accordingly.

If the build is successful, you can access the target application executable here:
./apps/target/test

10.1.2 Execute the Test Application

Copy the `./apps/target/test` file to the target. Execute it by specifying the number of CPU cores available on the target:

```
$ ./test -nt 4
```

The output may look like:

```
nt = 4

dataSize = 0x10000

Hotspot ip: 400AD1 : 4197073

numberOfThreads = 4
```

10.1.3 Configuration of Test Session

Use the test configuration to check the data returned by the Analysis Communication Agent. You can find the configuration file here:
./config.py

This is an example of a configuration file:

```
def ApolloLake(self):
        self.cores_number = 4                  # Number of cores the target platform
                                               # has
        self.testapp = 'test'                  # Name of test application binary.
                                               # Might require specifying full binary
                                               # path (depends on the target agent
                                               # capabilities).
        self.testapp_hotspot_ip = 0x400AD1     # The value is taken from the target
```

```
                                                    # application output.
        self.protocol_version = 6                   # Communication protocol version
```

10.1.4 Run Tests

To run tests, execute this command on your host system by specifying the IP address of the target system and the name of the configuration routine:
$ *python test.py* 192.168.1.1 *ApolloLake*

A successful validation generates output:

```
runTest (__main__.TerminateTest) ... ok
runTest (__main__.VersionTest) ... ok
runTest (__main__.SetupInfoTest) ... ok
runTest (__main__.SysConfigSizeTest) ... ok
runTest (__main__.SysConfigTest) ... ok
runTest (__main__.PlatformInfoTest) ... ok
runTest (__main__.InitNumDeviceTest) ... ok
runTest (__main__.BusyDriverTest) ... ok
runTest (__main__.SetEventConfigTest) ... ok
runTest (__main__.GetTscTest) ... ok
runTest (__main__.GetNumCoresTest) ... ok
runTest (__main__.CollectionTest) ... ok

----------------------------------------------------------------------
Ran 12 tests in 18.674s

OK
```

10.2 Test Scenarios

10.2.1 TerminateTest

This test validates basic communication functionality. The test initializes communicationby creating terminal/data channels (init). Then, it terminates the communication using the TERMINATE command and closes all channels. This functionality is common to all test case scenarios.

10.2.2 VersionTest

This test retrieves the driver version using the VERSION command.

10.2.3 SetupInfoTest

This test retrieves target agent setup information (e.g. ability to use NMI) using the GET_DRV_SETUP_INFO command.

10.2.4 SysConfigSizeTest

This test retrieves system configuration size using the COLLECT_SYS_CONFIG command.

10.2.5 SysConfigTest

This test retrieves system configuration using the GET_SYS_CONFIG command.

10.2.6 PlatformInfoTest

This test retrieves platform information using the GET_PLATFORM_INFO command.

10.2.7 InitNumDeviceTest

This test retrieves the number of profiling devices using the INIT_NUM_DEV command.

10.2.8 BusyDriverTest

This test retrieves driver busy status using the RESERVE command.

10.2.9 SetEventConfigTest

This test sends event configuration using the EM_GROUPS command.

10.2.10 GetTscTest

This test validates the TSC timestamp using the GET_NORMALIZED_TSC command.

10.2.11 GetNumCoresTest

This test retrieves the number of cores using the NUM_CORES command. It compares this value with the one specified in the test session configuration.

10.2.12 CollectionTest

This test verifies the entire collection flow using the mechanisms validated in the previous scenarios. It starts the collection of a single PMU event (CPU_CLK_UNHALTED.REF_TSC) for 5 seconds, retrieves the collected data, and terminates the channels.

Note: During the execution of CollectionTest, the test application must run on the target.

The test performs these actions:

- Validates the samples on the expected hotspot address
- Checks the name of the hotspot module
- Checks the overall samples count on the hotspot module

10.2.13 UncoreCollectionTest

This test is similar to CollectionTest, but also requests uncore IMC events to check for DRAM Memory Bandwidth observation. Depending on uncore support in the target agent, this test may be disabled.