ZeBu® Runtime Performance With zTune Application Note

Version V-2024.03-1, July 2024

SYNOPSYS®

Copyright and Proprietary Information Notice

© 2024 Synopsys, Inc. This Synopsys software and all associated documentation are proprietary to Synopsys, Inc. and may only be used pursuant to the terms and conditions of a written license agreement with Synopsys, Inc. All other use, reproduction, modification, or distribution of the Synopsys software or the associated documentation is strictly prohibited.

Destination Control Statement

All technical data contained in this publication is subject to the export control laws of the United States of America. Disclosure to nationals of other countries contrary to United States law is prohibited. It is the reader's responsibility to determine the applicable regulations and to comply with them.

Disclaimer

SYNOPSYS, INC., AND ITS LICENSORS MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Trademarks

Synopsys and certain Synopsys product names are trademarks of Synopsys, as set forth at https://www.synopsys.com/company/legal/trademarks-brands.html.
All other product or company names may be trademarks of their respective owners.

Free and Open-Source Licensing Notices

If applicable, Free and Open-Source Software (FOSS) licensing notices are available in the product installation.

Third-Party Links

Any links to third-party websites included in this document are for your convenience only. Synopsys does not endorse and is not responsible for such websites and their practices, including privacy practices, availability, and content.

www.synopsys.com

Feedback

Contents

Preface

This chapter has the following sections:

About This Book

The *ZeBu® Runtime Performance Analysis with zTune User Guide* describes runtime emulation performance analysis with zTune. It also describe the post processing steps and zTune limitations.

Contents of This Book

The *ZeBu® Runtime Performance Analysis with zTune User Guide* has the following chapters:

Chapter	Describes...
Introduction to zTune	Main Features of zTune
Data Collected by zTune	The types of data collected by zTune
Compile Time Options	Compile time options
zTune Runtime Environment	Runtime environments that support zTune
Post Processing	The steps that should be performed post processing
Limitations	zTune limitations

Feedback

Related Documentation

Document Name	Description
ZeBu User Guide	Provides detailed information on using ZeBu.
ZeBu Debug Guide	Provides information on tools you can use for debugging.
ZeBu Debug Methodology Guide	Provides debug methodologies that you can use for debugging.
ZeBu Unified Command-Line User Guide	Provides the usage of Unified Command-Line Interface (UCLI) for debugging your design.
ZeBu UTF Reference Guide	Describes Unified Tcl Format (UTF) commands used with ZeBu.
ZeBu Power Aware Verification User Guide	Describes how to use Power Aware verification in ZeBu environment, from the source files to runtime.
ZeBu Functional Coverage User Guide	Describes collecting functional coverage in emulation.
Simulation Acceleration User Guide	Provides information on how to use Simulation Acceleration to enable cosimulating SystemVerilog testbenches with the DUT
ZeBu Verdi Integration Guide	Provides Verdi features that you can use with ZeBu. This document is available in the Verdi documentation set.
ZeBu Runtime Performance Analysis With zTune User Guide	Provides information about runtime emulation performance analysis with zTune.
ZeBu Custom DPI Based Transactors User Guide	Describes ZEMI-3 that enables writing transactors for functional testing of a design.
ZeBu LCA Features Guide	Provides a list of Limited Customer Availability (LCA) features available with ZeBu.
ZeBu Synthesis Verification User Guide	Provides a description of zFmCheck.
ZeBu Transactors Compilation Application Note	Provides detailed steps to instantiate and compile a ZeBu transactor.
ZeBu zManualPartitioner Application Note	Describes the zManualPartitioner feature for ZeBu. It is a graphical interface to manually partition a design.
ZeBu Hybrid Emulation Application Note	Provides an overview of the hybrid emulation solution and its components.

Typographical Conventions

This document uses the following typographical conventions:

To indicate	Convention Used
Program code	`OUT <= IN;`
Object names	`OUT`
Variables representing objects names	`<sig-name>`
Message	Active low signal name '<sig-name>' must end with _X.
Message location	`OUT` <= IN;
Reworked example with message removed	`OUT_X` <= IN;
Important Information	**NOTE:** This rule...

The following table describes the syntax used in this document:

Syntax	Description
[ ] (Square brackets)	An optional entry
{ } (Curly braces)	An entry that can be specified once or multiple times
| (Vertical bar)	A list of choices out of which you can choose one
... (Horizontal ellipsis)	Other options that you can specify

Synopsys Statement on Inclusivity and Diversity

Synopsys is committed to creating an inclusive environment where every employee, customer, and partner feels welcomed. We are reviewing and removing exclusionary language from our products and supporting customer-facing collateral. Our effort also includes internal initiatives to remove biased language from our engineering and working environment, including terms that are embedded in our software and IPs. At the same time, we are working to ensure that our web content and software applications are usable to people of varying abilities. You may still find examples of non-inclusive language in our

software or documentation as our IPs implement industry-standard specifications that are currently under review to remove exclusionary language.

1

Introduction to zTune

zTune is an emulation performance analysis tool. When a design is compiled with **zTune** profiling enabled, **zTune** generates hardware and software profiles for the design that can be viewed simultaneously using a common web browser. It helps users to identify emulation performance bottlenecks.

The main features of **zTune** include:

- Profiling clock frequencies, clock stopping activity, and software activity
- Profiling data analysis using an interactive GUI

Profiling does not have a significant effect on the overall system performance.

To generate **zTune** profiling data, the design must be compiled with the "`profile -xtors true`" option in the UTF file. The user must also add a few **zTune** API routines (see Introduction to zTune) to their code to configure, and activate the profiling during runtime.

When performance profiling is enabled, a directory is generated during emulation to save the profiling data. After emulation, the user can pass this directory as an argument to **zTune** for post processing and analyze the emulation performance using the web browser of their choice to access the **zTune** GUI (see Post Processing).

2

Data Collected by zTune

There are many ways to measure emulation performance. Most commonly, the performance is measured by comparing the progress of simulated time to wall clock time. The simulated real time depends on the DUT controlled clocks and frequencies. This chapter describes the types of data collected by **zTune**.

For more information, see the following subsections:

- Controlled Clocks
- Clock Frequency
- Clock Stopping Activity

Controlled Clocks

Emulation performance is determined by clock frequencies in the design. During emulation, some components of the design can stop the clocks (Controlled Clocks) and lower the average clock frequency.

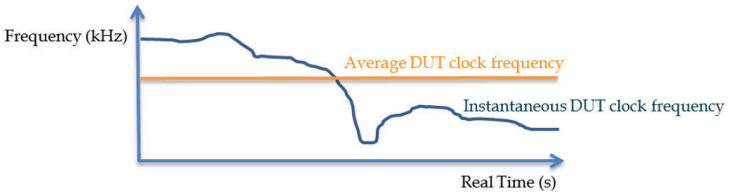
Controlled clocks can be stopped at runtime for multiple reasons. Transactors, including co-simulation transactors such as `C_COSIM`, may stop controlled clocks through a clock controller instance. Controlled clocks may also be stopped by ZeBu features such as Power Management (POMA) or turned on. **zTune** collects data to help you identify the component responsible for stopping the controlled clocks.

Clock Frequency

The average frequency and instantaneous frequency of the DUT's controlled clocks are important parameters to analyze runtime performance. Instantaneous frequency is the clock frequency measured in a small time interval, for example 10 ms. This time interval

is called the sampling rate. The following diagram displays average and instantaneous frequencies collected by the profiler:

Figure 1 Frequency Monitoring



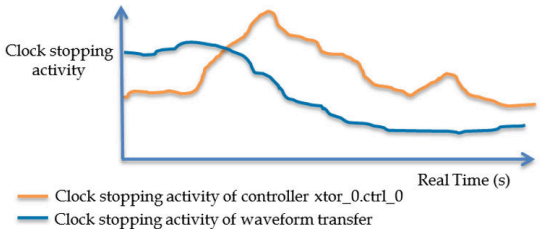
The instantaneous frequency helps identify the time window when a controlled clock slowdown occurs.

Clock Stopping Activity

Clock stopping slows down the emulation time and results in lower clock frequencies. Hence, it is important to determine the reason for the frequency slowdown. Clock stopping analysis determines the clock controller that stopped the clocks and the moment when the clocks stopped.

This requires obtaining the instantaneous stop status of each clock controller. The instantaneous stop status is defined as the number of driver clock cycles that the `readyForClock` signal is 0 during a small time interval. The following graph helps identify the origin of the clock stopping requests.

Figure 2 Monitoring Clock Stopping Activity



3

Compile Time Options

To include **zTune** transactional profiler, add the following command in the UTF file while compiling the design:

```
profile -compile true -xtors true -xtors_params
 {ZEMI_GLOBAL|ZEMI_DPI|ZEMI_GLOBAL_DPI|LEGACY}
```

where,

- `ZEMI_GLOBAL`: Profile global (`sampling/internal/dpi/b2b`) clock stopping for ZEMI transactors
- `ZEMI_DPI`: Profile per DPI clock stopping for ZEMI transactors
- `ZEMI_GLOBAL_DPI`: Profile both global and per DPI clock stopping for ZEMI transactors. This is the default value.
- `LEGACY`: Legacy profiling (per transactor `zceiClockControl`)

By default, the `profile` command enables profiling for:

- SW stack
- ZCEI controlled clocks
- ZCEI clock control macros

4

zTune Runtime Environment

zTune profiling is supported by **ZeBu Runtime Control Interface (zRci)**.

To start **zTune** profiling in **zRci**, use the following command:

```
start_zebu -ztune <some_dir>
```

The following options can be used to enable **zTune** operations:

```
ztune    # zTune operations.
       ztune -start [path]
       ztune -stop
       ztune -init
       ztune -create_event_marker [name]
       ztune -config <option> [value]
       Options:
              output_path <path>
              csv [on|off|1|0]
              hw_profiler [on|off|1|0]
              hw_poll_delay [delay in us]
              port_tracking [on|off|1|0]
              real_time_profiler [on|off|1|0]
              cpu_profiler [on|off|1|0]
              sw_sampling_period [period in ms]
              system_marker_auto_creation [on|off|1|0]
              user_markers_max [max]
```

Where,

- `[-start [<path>]]`: Starts data recording
- `[-stop]`: Stops data recording
- `[-init]`: Initializes **zTune**. It must be called before `start_zebu`
- `[-create_event_marker [<name>]]`: Creates an event marker
- `[-config <option> [<value>]]`: Fetches or sets the **zTune** configuration option

5

Post Processing

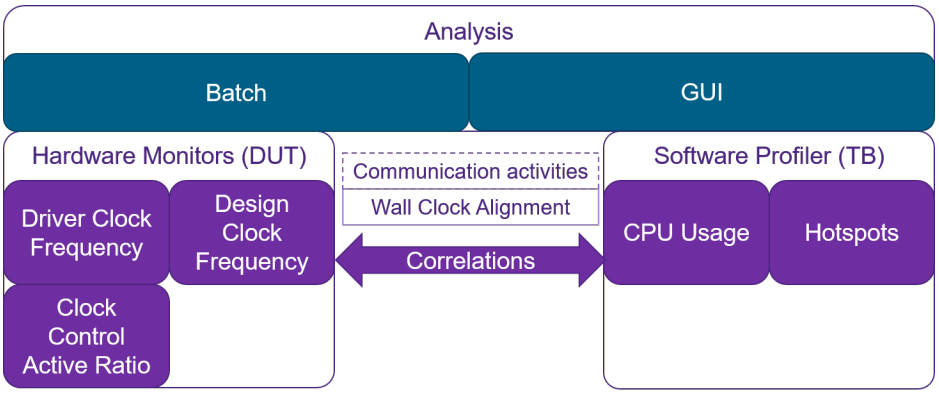
zTune is available in *$ZEBU_ROOT/bin*

```
$ZEBU_ROOT/bin/zTune <path to profiling database>
```

The profiling data is saved in the directory specified by the `output_path` config option of the **zTune** command in **zRci**. The contents of this folder are analyzed by **zTune**. It displays profiling data graphically through a GUI or in a command line mode.

The following figure displays data analysis performed by **zTune**.

Figure 3 Data Analysis in zTune



This chapter consists of the following sections:

- zTune Command-Line Options
- zTune GUI
- GUI Views
- System Stoppers Information
- System Stoppers Information
- zTune Text-Based UI

zTune Command-Line Options

The various command-line options covered in this section are as follows:

Command Line for Help

Syntax:

```
zTune [--help]
zTune [--id]
```

where,

- `--help`: Displays the help messages and exits

Command line for GUI

Syntax:

```
zTune [--filter_clockControl_percentage <integer>]
 [--filter_clockControl_number <integer>] [--cycles_scale] <database>
```

where,

- `--filter_clockControl_percentage <integer>`: Displays clock controls that are less than `<integer>` [Default:95]
- `--filter_clockControl_number <integer>`: Displays `<integer>` clock controls [Default 20]
- `--cycles_scale`: Displays clock cycles
- One positional argument, that is, `<database>`, must be used with `zTune` to point to the profile database.

Command Line for Interactive Text Mode

Syntax:

```
zTune --interactive <database>
```

where,

- `--interactive`: Specifies interactive mode
- One positional argument, that is, `<database>`, must be used with `zTune` to point to the profile database.

Command Line for Script Mode

Syntax:

```
zTune --command_file <filename> <database>
```

where,

- `--command_file <filename>`: Executes commands in the input command file
- One positional argument, that is, `<database>`, must be used with `zTune` to point to the profile database.

Command Line to Print Information About Threads

Syntax:

```
zTune --process <count> <database>
```

where,

- `--process <count>`: Lists the specified number of threads sorted by CPU time
- One positional argument, that is, `<database>`, must be used with `zTune` to point to the profile database.

Command Line to Print Information About Functions

Syntax:

```
zTune --function incl|excl <thread_id> <count> <database>
```

Feedback

where,

- `--function incl|excl <thread_tid> <count>`: Lists functions sorted by inclusive or exclusive time in thread
- One positional argument, that is, `<database>`, must be used with `zTune` to point to the profile database.

Common Options

Syntax:

```
zTune [--port PORT] [--host HOST] [--log LOG]
```

where,

- `--port <port>`: Specifies the port that zTune listens on. The default port is 1106
- `--host <host>`: Specifies the host that zTune listens on. The default host is `localhost`.
- `--log <filename>`: Redirects output into a file

zTune GUI

The **zTune** post-processing tool requires a modern web browser and Python 2.7 (provided in the ZeBu release) to display the results in the GUI.

Launching zTune GUI

To launch **zTune** after emulation runtime, perform the following steps:

1. Start a web server using the following **zTune** script:

   ```
   $ zTune <some_dir>/profile_dir
   ```

   The **zTune** script creates a web server on the machine where the script was running and displays the address on the Unix terminal from where **zTune** was launched from:

   ```
   Loading profile database...
   Profile database is load successfully.
   Listening on http://vginthw141:1106
   Web server activities will be logged in collect_web.log
   ```

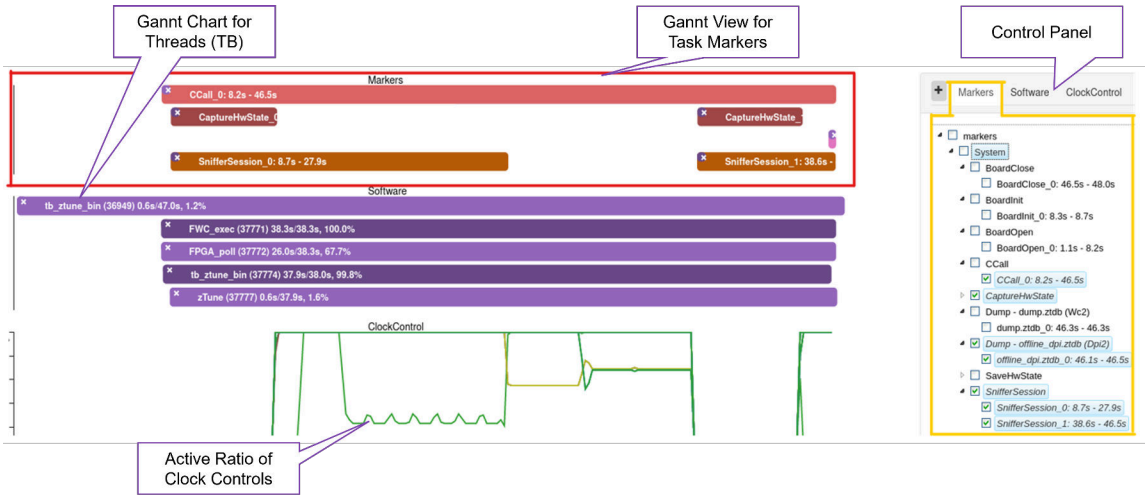
2. To access data, open a web browser in Windows or Linux and enter this web server address to launch the **zTune** GUI.

   http://vginthw141:1106

Feedback

The **zTune** GUI home page is displayed with the information from the profile directory.

Figure 4 zTune Initial GUI



The left-hand panel of the **zTune** GUI is divided into four parts:

- Software: Displays the active threads
- Clock Controls: Displays the active ratio of clock controls
- Clocks: Displays the frequency of each primary clock in the design
- Markers: Displays the main ZeBu activities such as waveform capture, SVA, and so on

All the 4 parts share the same X-axis. For more information, see Overall View.

On the right-hand side, beside the graphic representation, you can see a control panel with the following tabs:

- **Software**
- **Clock**
- **ClockControl**
- **Markers**

Each tab controls the content of the panelof the same.

The visible markers, threads, clocks, and clock controls can be selected from the list of each panel.

Feedback

GUI Views

The **zTune** GUI presents the following levels of software profile views:

- Overall View
- Thread View
- Function View (C/C++ testbench)

You can also represent markers (predefined and user-defined) in both Overall and Thread profile views. For more information about markers, see Markers.

Overall View

The initial **zTune** GUI view is also called Overall view. It is a graph with real emulation time on the x-axis. The horizontal bars represent the time during which processes and threads are running.

Figure 5 Overall View

Feedback

You can obtain the following information from the overall view:

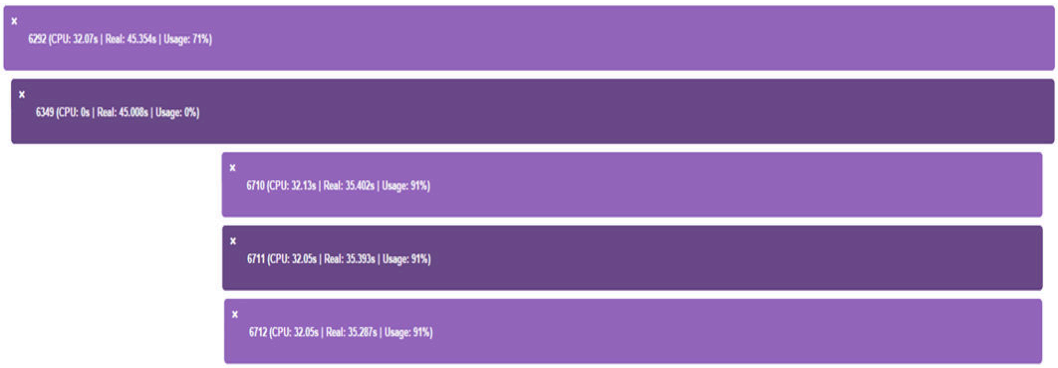
- **Process ID**: The process ID is displayed on the left side of the view, as displayed in the following figure:

Figure 6 Process ID

You can view the command used to launch this process if you hold the pointer over this process ID.

- **Thread Bars**: The thread bars display when the process threads are launched and finished. You can find the following information displayed on the bars in the GUI:
  - Thread ID
  - CPU Time (in seconds)
  - Real Time (in seconds)
  - CPU usage (in percentage)

Figure 7 Threaded Bars



Markers

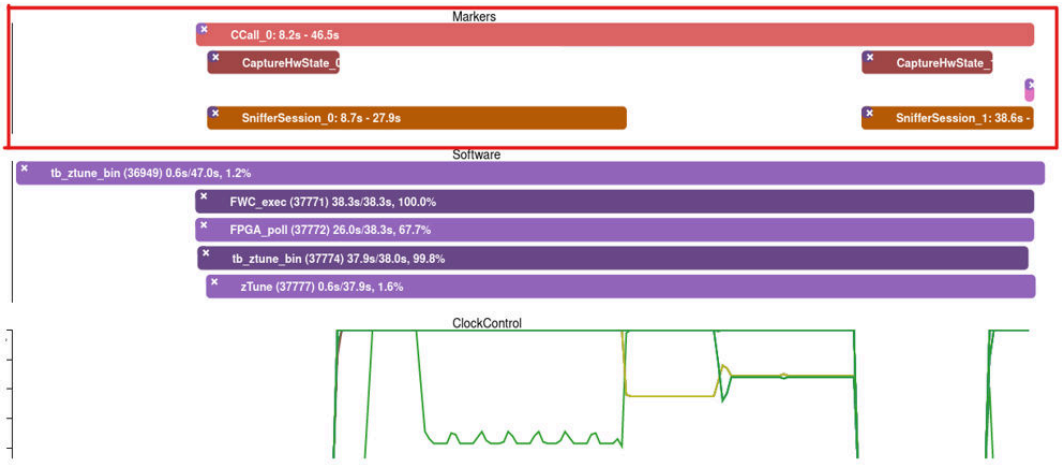
In the following zTune GUI, marker Gannt view represent predefined task types stored in the task database. Each task type represents the type of runtime activity. For every given

Feedback

task, the events happen linearly without overlapping. The task is mapped to a task map that enables you to retrieve the current task from different function calls in the runtime API. Some of the defined task types are as follows:

- `BoardClose`
- `BoardInit`
- `BoardOpen`
- `CCall`
- `CaptureHwState`
- `Checkpoint`
- `Dump`
- `Reopen`
- `RestoreHwState`
- `SaveHwState`
- `SnifferSession`
- `SnifferCreateFrame`
- `Sva`

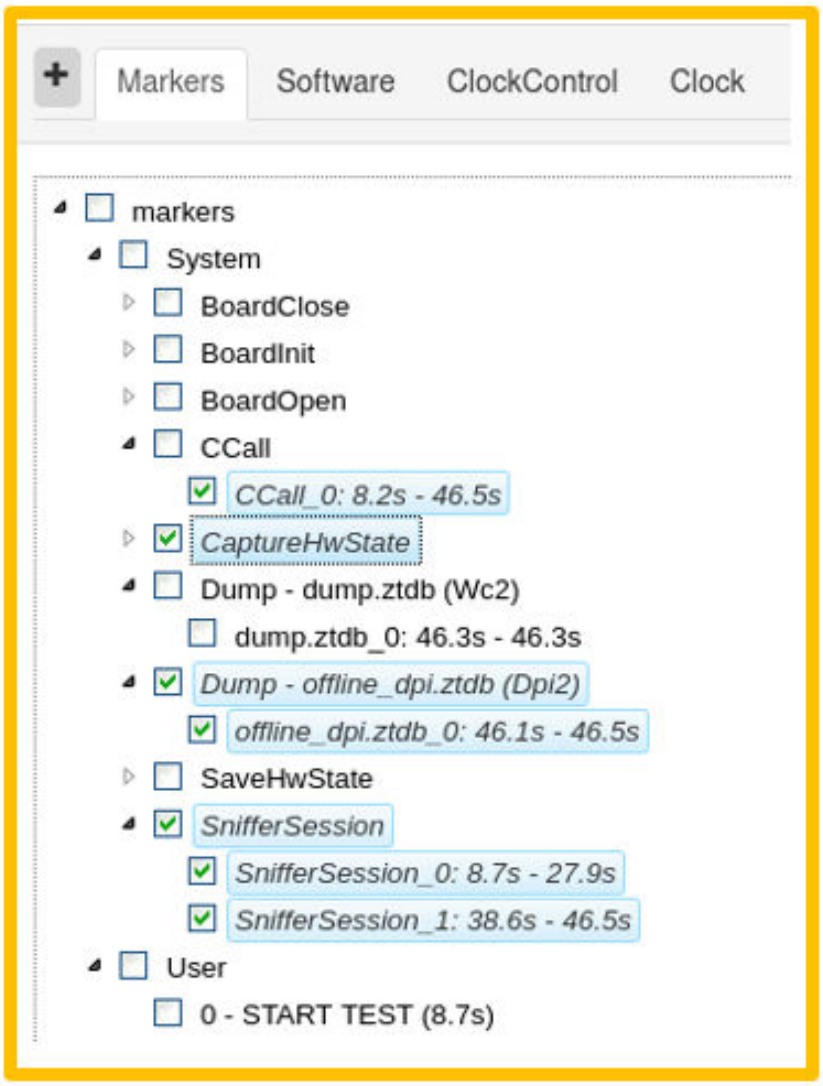
Figure 8 Representing Markers in a zTune GUI View



Feedback

You can select the task to be displayed and check the time in the control panel as shown in the following figure:

Figure 9 Task Markers Selection in zTune GUI

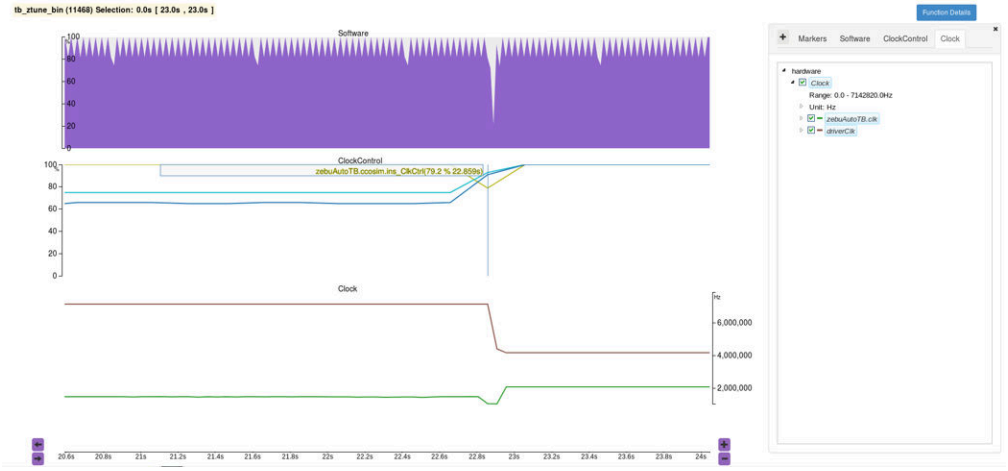


Thread View

Click a thread bar in the overall view to open the thread view. The Thread view is a graph with real emulation time on the horizontal axis. It displays a graphical representation of the CPU usage during the lifetime of the selected thread.

Feedback

Figure 10 Thread View



You can select or deselect any process by clicking any process. If you double-click any process, it opens “**Thread View**”.

You can select any thread in the design listed in the **Software** tab of the control panel by selecting the check box next to the “thread name/number”.

Figure 11 Software Profile Configuration Page

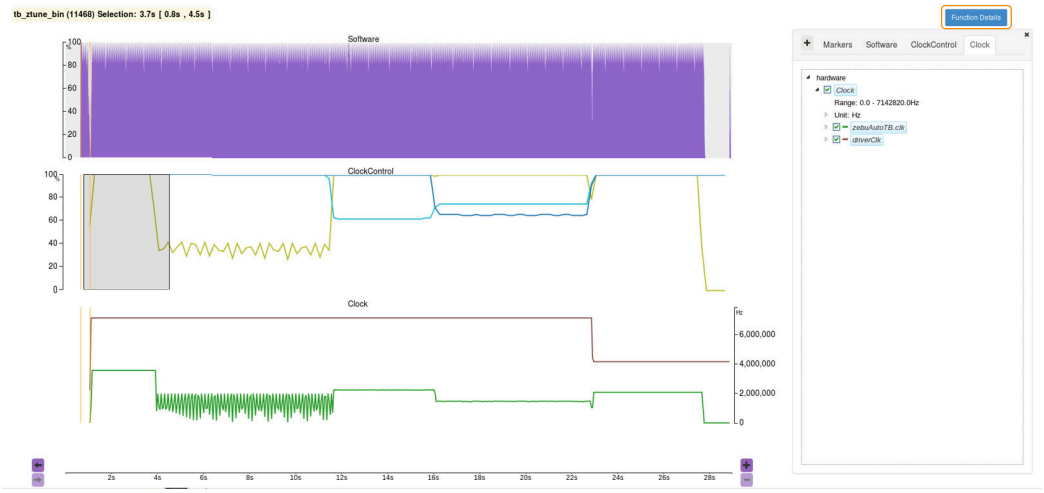
Function View (C/C++ testbench)

The function view displays the CPU usage of individual C/C++ functions. You can open the Function View of a thread by selecting a range in the Thread View and then clicking Function Details in the same view.

Feedback

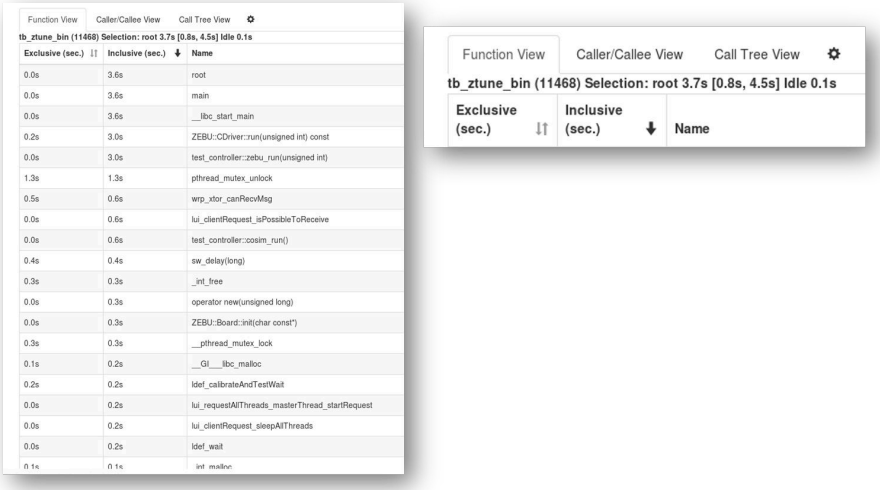
See the following figure:

Figure 12 Selecting a Range in Thread View



This opens the Function View for the selected range as displayed in the following figure:

Figure 13 Function View



Note:

If you do not select any range, the function view displays the entire lifetime of the selected thread.

Feedback

You can select the range according to the emulation profile data to find out the behavior of any process thread for the time during which performance degrades. The following types of views are available:

- Function View (Figure 13)
- Caller/Callee View
- Call Tree View

You can switch to the required view by clicking the respective view name in the Function View window.

Figure 14 Caller/Callee View

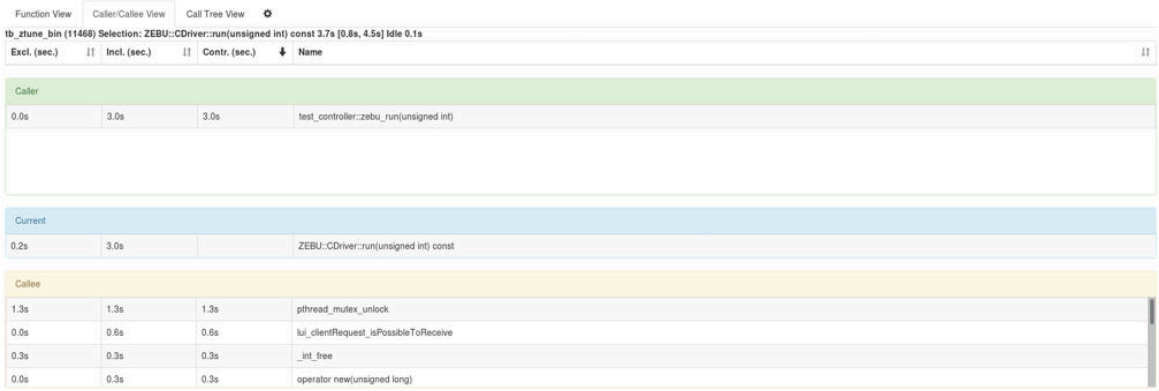
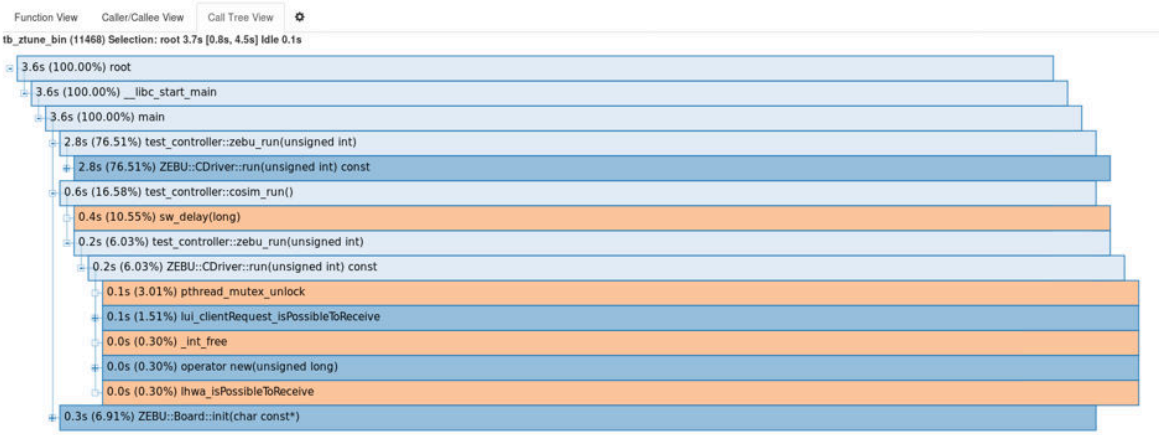


Figure 15 Call Tree View



Feedback

System Stoppers Information

In ZeBu Server 4 and 5, due to a change in clock generation and distribution, only one system stop signal is available for each unit:

- For FWC stops (one per unit with FWC IP): `system_stoppers.fwc_stop.Ui`

For transactional memory stops (`memx`) (one signal per unit in the clock control graph): `system_stoppers.memx_stop.Ui`.

zTune Text-Based UI

zTune can generate textual output using the following methods:

- By using zTune interactively. The command is as follows:

  ```
  zTune --interactive profile_dir
  ```

- By providing a script containing commands. The command is as follows:

  ```
  zTune --command_file script.ztune profile_dir
  ```

- By using command line options. The command is as follows:

  ```
  zTune -proc 0 profile_dir
  ```

  Note:

  This method is not available for all commands.

This section covers commands that enable:

- Printing Function Execution Time
- Printing Thread Execution Time
- Printing Hardware Metadata
- Printing Samples for a Hardware Component
- Printing Information About Functions
- Printing Markers
- Printing Information About Hardware

Feedback

Printing Function Execution Time

To print functions sorted by "incl"/"excl" execution time in interactive mode, use the following command:

```
func [incl|excl] <tid|pid> <count>
```

The corresponding command line option is as follows:

```
--function [incl|excl] <tid> <count>
-func [incl|excl] <tid> <count>
```

where,

- “incl” is the execution time of a function, including the time consumed by functions called directly or indirectly.
- "excl" is the time consumed by a function, excluding the time consumed by the functions called directly or indirectly.
- tid/pid should be a valid process thread ID. If 0 or an invalid ID is given, the whole thread/process data is displayed.
- count should be a positive integer number – the number of functions.

Example:

```
[zTune]$ func incl 2878 10
         pid: 2878_19390_1650052646
excl(s)
incl(s)
name
126.160
801322.030
root
0.000
801195.870
main
0.000
801195.870
__libc_start_main
0.000
793822.370
Test_Basic::run()
165382.250
791700.310
DutDriver_Ccosim::run(unsigned long)
0.000
722569.200
Test::runBasic(unsigned long)
0.000
288090.530
```

Feedback

```
DutDriver_Ccosim::load(unsigned long, std::array<unsigned int, 3ul>)
0.000
216073.820
DutDriver_Ccosim::idle(unsigned long)
201253.760
201253.760
pthread_mutex_unlock
188307.960
188307.960
pthread_mutex_lock
```

Printing Thread Execution Time

The `proc` command lists the specified number of threads or processes sorted by execution time. The count should be non-negative integer; if `0` is given, the whole thread or process data is displayed.

The corresponding command line option is as follows:

Syntax:

```
proc <count>
```

Example:

```
[zTune]$ proc 10
id
cpu(s)
real(s)
2878
80.1s
92.3s
3600
1.3s
79.7s
3272
0.3s
92.0s
3271
0.1s
92.0s
3539
0.1s
80.8s
3273
0.0s
92.0s
3274
0.0s
92.0s
3279
0.0s
```

Feedback

```
92.0s
3281
0.0s
92.1s
3283
0.0s
92.0s
```

Printing Hardware Metadata

The `hardware --meta` command captures hardware metadata (reference frequency, sample period, monitored hardware components, and its indexes).

Syntax:

```
report_hardware --meta
```

Example:

```
[zTune]$ report_hardware --meta
hardware hw_top.clockGen12.clock 0
hardware hw_top.clockGen11.clock 1
hardware hw_top.clockGen10.clock 2
hardware hw_top.clockGen09.clock 3
hardware hw_top.clockGen08.clock 4
hardware hw_top.clockGen07.clock 5
hardware hw_top.clockGen06.clock 6
hardware hw_top.clockGen05.clock 7
hardware hw_top.clockGen04.clock 8
hardware hw_top.clockGen03.clock 9
hardware hw_top.clockGen15.clock 10
hardware hw_top.clockGen02.clock 11
hardware hw_top.clockGen01.clock 12
hardware hw_top.clockGen14.clock 13
hardware hw_top.clockGen00.clock 14
hardware hw_top.clockGen13.clock 15
hardware driverClk 16
hardware hw_top.dut_ccosim.zcei_stopn 17
hardware hw_top.ts_runman_0.zcei_stopn 18
hardware hw_top.ts_runman_0.zcei_stopn_0 19
hardware hw_top.ts_runman_0.zcei_stopn_1 20
hardware hw_top.ts_runman_0.zcei_stopn_2 21
hardware hw_top.ts_runman_0.zcei_stopn_3 22
[...]
hardware hw_top.ts_runman_10.zcei_stopn_13 192
hardware hw_top.ts_runman_10.zcei_stopn_14 193
hardware system_stoppers.fwc_stop.U0 194
```

Feedback

Printing Samples for a Hardware Component

The `hw` index command captures sample information for hardware component index. It retrieves the index from the `hw` meta command.

Syntax:

```
[-h] --id ID
```

Example:

```
[zTune]$ report_hardware --id 17
12.56s 100
91.96s 100
```

Printing Information About Functions

To find the functions, use the `find_functions` command:

Syntax:

```
find_functions --name <name> [--limit <limit>]
                   [--time-unit <absolute,seconds,cycle-counter>]
                   [--ref-clk <clock>]
                   [--sort <inclusive|exclusive>]
                   [--csv <filename>]
                   [--interval [<begin>:<end>|<begin>:|:<end>]]
```

where,

Optional arguments that can be used with `find_functions` are as follows:

- `--name <name>`: Function name (`regexp`)
- `--limit <limit>`: Maximum number of functions to display, when `<limit>` is 0, display all threads
- `--time-unit [absolute|seconds|cycle-counter]`: Time scale to display

  Note:

  `--ref-clk` is mandatory with `--time-unit=cycle-counter`
- `--ref-clk <clock>`: Reference clock to which to display the cycle count (`regexp`)
- `--sort [inclusive|exclusive`: Sorts elements and accepts the minimum valid string
- `--csv [<filename>]`: Report in `csv` format. The default option is `stdout`.
- `--interval [<begin>:<end>|<begin>:|:<end>]`

Feedback

Example 1:

```
$ find_functions -n zebu --csv
```

This command displays Function Name, Thread ID, Thread Name, Time Range, Inclusive Time, Exclusive Time, Callers.

```
test_controller::zebu_run(unsigned
 int),2367,tb_ztune_bin,10.06s--46.96s,22.99s,0.0s,"test_controller::cosi
m_run(),main"
```

Example 2:

```
[zTune]$ find_functions --name DutDriver_Ccosim --interval 20: --time-
unit seconds --sort inclusive

-----------------------------------------------------------------
Function Name | Thread ID | Thread Name | Time Range | Inclusive Time |
 Exclusive Time | Callers
-----------------------------------------------------------------
DutDriver_Ccosim::run(unsigned long) | 2878 | testbench | 20.0s--92.01s |
 71.89s | 15.02s |
DutDriver_Ccosim::reset(unsigned long),DutDriver_Ccosim::idle(unsigned
 long),DutDriver_Ccosim::load(unsigned long,
 std::array<unsigned int, 3ul>),DutDriver_Ccosim::write(unsigned
 long),DutDriver_Ccosim::read(unsigned long)
DutDriver_Ccosim::load(unsigned long, std::array<unsigned int,
 3ul>) | 2878 | testbench | 20.0s--91.55s | 26.06s | 0.0s |
 Test::runBasic(unsigned long)
DutDriver_Ccosim::idle(unsigned long) | 2878 | testbench | 20.0s--92.01s
 | 19.71s        | 0.0s      | Test::runBasic(unsigned long)
DutDriver_Ccosim::read(unsigned long) | 2878 | testbench | 20.0s--91.82s
 | 13.18s    | 0.0s      | Test::runBasic(unsigned long)
DutDriver_Ccosim::reset(unsigned long)| 2878 | testbench | 20.0s--89.65s
 | 6.48s     | 0.0s       | Test_Basic::run()
DutDriver_Ccosim::write(unsigned long)| 2878 | testbench | 20.0s--90.31s
 | 6.47s | 0.0s | Test::runBasic(unsigned long)
```

Printing Markers

To print the list of markers, use the `report_markers` command.

Syntax:

```
report_markers [-h] --name NAME [--interval BEGIN:END|BEGIN:|:END]
                     [--time-unit {absolute,seconds,cycle-counter}]
                     [--ref-clk REF_CLK] [--csv [=STDOUT|FILENAME]]
                     [--type {all,event,task}]
```

Feedback

where,

`report_markers` supports the following optional arguments:

- `-h, --help`: Shows the online help and exits
- `--name NAME, -n NAME`: Selects the marker data name

  Note:

  Regular expressions are supported.
- `--interval BEGIN:END|BEGIN:|:END, -i BEGIN:END|BEGIN:|:END`: Selects a time interval
- `--time-unit {absolute,seconds,cycle-counter}, -t {absolute,seconds,cycle-counter}`: Provides scale of time to display
- `--ref-clk REF_CLK, -r REF_CLK`: Sets reference clock to which to display the cycle count (`regexp`)
- `--csv [=STDOUT|FILENAME]`: Generates report in the `csv` format (prints by default on `stdout`)
- `--type {all,event,task}`: Filters results on marker type

Example:

```
[zTune]$ report_markers -n .*
--------+---------------------
Time[s] | Name
--------+---------------------
8.73    | START TEST
16.21   | START FULL SPEED RUN
18.61   | START PREEMPTED RUN
27.91   | START DPI PHASE
27.91   | ONE DPI
32.22   | MULTIPLE DPI
38.63   | END DPI PHASE
46.04   | START FWC DUMP
46.27   | END FWC DUMP
46.27   | END TEST


------------------+---------------+------------
Name              | Begin Time[s] | End Time[s]
------------------+---------------+------------
BoardOpen_0       | 1.05          | 8.19
CCall_0           | 8.22          | 46.54
BoardInit_0       | 8.31          | 8.72
SnifferSession_0  | 8.73          | 27.91
CaptureHwState_0  | 8.74          | 14.78
SaveHwState_0     | 14.78         | 16.18
```

Feedback

```
SnifferSession_1    | 38.63          | 46.51
CaptureHwState_1    | 38.65          | 44.63
SaveHwState_1       | 44.64          | 46.02
offline_dpi.ztdb_0  | 46.10          | 46.53
dump.ztdb_0         | 46.27          | 46.30
```

Printing Information About Hardware

Syntax:

```
report_hardware [-h] (--meta | --id ID | --name NAME) [--raw]
                        [--interval BEGIN:END|BEGIN:|:END]
                        [--time-unit {absolute,seconds,cycle-counter}]
                        [--ref-clk REF_CLK] [--csv [=STDOUT|FILENAME]]
                        [--category {Clock Control,Clock}]
                        [--value-unit {%,Hz,kHz}]
                        [--precision TIME(s|ms|us) | POINTS]
```

where,

`report_hardware` supports the following arguments:

- `-h, --help`: Shows the help messages and exits
- `--meta`: Prints hardware metadata

  Note:

  This is only compatible with `--category`. Other options are ignored, if present.

- `--name NAME, -n NAME`: Hardware data name. This is a mandatory argument.

  Note:

  This argument supports regular expressions.

- `--id ID`: Hardware data id
- `--raw`: Prints unprocessed hardware sample data.

  Note:

  This is only compatible with `--id/--name`. Other options are ignored, if present.

- `--category Clock Control|Clock`: Category of hardware data
- `--time-unit [absolute|seconds|cycle-counter], -t {absolute,seconds,cycle-counter}`: Scale of time to display.

Note:

`--ref-clk` is mandatory with `--time-unit=cycle-counter`

- `--value-unit [Hz|kHz|%], -v {%,Hz,kHz}`
- `--ref-clk REF_CLK, -r REF_CLK`: Reference clock for which the cycle count is displayed. Use this option only when `--time-unit cycle-counter` is used.

  Note:

  This argument supports regular expression.
- `--precision TIME(s|ms|us) | POINTS, -p TIME(s|ms|us) | POINTS`: Specifies interval between each sample or number of samples displayed (minimum 2)
- `--csv [=STDOUT|FILENAME]`: Displays output / data in the `csv` format. The default format is *STDOUT*.
- `--interval BEGIN:END|BEGIN:|:END, -i BEGIN:END|BEGIN:|:END`: Displays data only on specified interval.

Consider the following examples.

- `--interval 10:20` between 10 and 20 time units (as per --time unit)
- `--interval 10`: between 10 time units and end of data
- `--interval :30` between beginning of data and 30 time units

Examples

```
 $report_hardware --name=driverClk --csv
     Time[s],driverClk[Hz]
     0,0
     0.2,1000
     0.4,10000
     0.6,10000
     ...

 $report_hardware --name=ccosim*
Time[s]   hw_top.ccosim_clk[Hz]  hw_top.ccosimDriver.readyForCclock[Hz]
     0           0                       0
     1           1000                    30
     2           3000                    100
```

6

Limitations

zTune has the following limitations:

- Automated analysis of results is not supported.
- Hardware and software message exchange activity is not collected.
- Sampling rate cannot be configured.

  Note:

  The firmware level logic analyzer is no longer available in ZeBu Server 4 and thus no `system_stoppers.analyzer_stop` signal is seen.