Analog Devices, Inc.; Antmicro

Zeppelin - Zephyr AI Profiling Library - documentation

2025-08-28

CONTENTS

CHAPTER

ONE

INTRODUCTION

Zephyr Profiling Library (ZPL), or Zeppelin for short, is a library which enables capturing and reporting runtime performance metrics, for the profiling and detailed analysis of Zephyr applications, with a special focus on applications running AI/ML inference workloads.

This documentation describes the following aspects of Zeppelin and the associated projects:

- *Zeppelin project* - provides general information on Zeppelin repository, describes how to build, test and use the profiling middleware
- *Zeppelin configuration* - describes build-time and runtime configuration of Zeppelin library
- *Memory profiling* - describes memory profiling, along with memory events
- *Tracing code scopes* - describes the use of tracing code scopes
- *CTF to TEF conversion* - describes how traces are converted and processed
- *Zeppelin Trace Viewer* - describes the tool for trace visualization

CHAPTER

TWO

ZEPPELIN PROJECT

Zeppelin is implemented as a West module. The Zeppelin repository consists of the following elements:

- Zeppelin library
- Custom configurations of boards based on MAX78002 and MAX32690 - for testing purposes
- Sample applications, which also serve as integration tests
- Unit tests
- Patches to Zephyr RTOS

2.1 Initializing the workspace

First, make sure all dependencies required by Zephyr RTOS are installed - follow the Getting started guide.

Secondly, create a workspace and clone the Zeppelin repository:

```
mkdir workspace && cd workspace
git clone --recursive git@github.com:antmicro/zeppelin.git
cd zeppelin
```

Then, install `west` and additional dependencies listed in project's `requirements.txt` with `pip`:

```
pip install -r requirements.txt
```

Next, initialize the workspace using West. To do that, run the following command:

```
west init -l .
```

Download, patch and prepare the project sources using the following commands:

```
west update
west patch apply
west zephyr-export
west packages pip --install
```

For testing without hardware in the loop, download Renode portable and add the download path to `PATH` environment variable:

```
wget https://builds.renode.io/renode-latest.linux-portable-dotnet.tar.gz
mkdir renode-portable
tar --strip-components=1 -C ./renode-portable -xvf renode-latest.linux-portable-
↪dotnet.tar.gz
export PATH=$(pwd)/renode-portable:$PATH
```

Finally, download Zephyr SDK:

```
west sdk install
```

2.2 Running a sample project with Zeppelin

To collect traces and visualize them using Zeppelin Trace Viewer, you can run *a simple demo with gesture recognition*, based on the data from an accelerometer in a Renode simulation. The default *configuration* in this demo collects traces along with all possibile additional information, like memory usage, die temperature, inference statistics, and more. One UART provides logs from the application, whereas the other UART returns CTF traces.

To build the demo, run:

```
west build -p -b stm32f746g_disco/stm32f746xx samples/demo
```

To run it in a Renode simulation, run:

```
python ./scripts/run_renode.py \
    --repl ./samples/demo/boards/stm32f746g_disco_lis2ds12.repl \
    --sensor i2c1.lis2ds12 \
    --sensor-samples ./samples/common/data/magic_wand/magic_wand.data \
    --trace-output trace.ctf \
    --timeout 10
```

This demo will for 10 seconds until a timeout is reached. After this time, CTF traces returned over secondary UART will be stored in `trace.ctf`.

Note

For trace collection on actual hardware, refer to *Trace collection*.

The trace needs to be converted to the TEF format, so that it can be loaded in Zeppelin Trace Viewer.

For that purpose, run:

```
west zpl-prepare-trace ./trace.ctf --tvm-model-path samples/common/tvm/model/
↪magic-wand-graph.json -o ./tef_tvm_profiler.json
```

The part `--tvm-model-path` is an input argument with the path to a TVM model graph, which is used to introduce additional model data to the TEF trace file metadata.

To get an overview of the traces, load the output `tef_tvm_profiler.json` file in Zeppelin Trace Viewer.

2.3 Customizing and using Zeppelin

Zeppelin can be enabled by y-selecting the `CONFIG_ZPL` symbol in the project configuration file. To initialize Zeppelin in a runtime, use the `zpl_init()` function defined by the `zpl/lib.h` header. You can enable various Zeppelin components by using Kconfig and runtime configuration, as described in following sections.

2.3.1 Configuration

The library can be configured both during building and during a run on a device. To find out how to configure the library and how to add new configurations, check *Zeppelin configuration*.

2.3.2 Trace collection

To enable Zeppelin tracing support, the user should enable the symbol `CONFIG_ZPL_TRACE` in Kconfig. You can then select one of the following formats:

- Plaintext format, by y-selecting `CONFIG_ZPL_TRACE_FORMAT_PLAINTEXT`
- Common Trace Format (CTF), by y-selecting `CONFIG_ZPL_TRACE_FORMAT_CTF`

You can choose how the traces will be delivered to the host PC by selecting one of the available tracing backends:

- UART, by y-selecting `CONFIG_ZPL_TRACE_BACKEND_UART`
- USB, by y-selecting `CONFIG_ZPL_TRACE_BACKEND_USB`
- Debugger, by y-selecting `CONFIG_ZPL_TRACE_BACKEND_DEBUGGER`
- Renode's simulated trivial UART, by y-selecting `CONFIG_ZPL_TRACE_BACKEND_TRIVIAL_UART`

Depending on the tracing backend used, the following commands can be used for trace capture.

UART

- Config option - `CONFIG_ZPL_TRACE_BACKEND_UART`
- Command:

```
west zpl-uart-capture [-h] serial_port serial_baudrate output_path

Capture traces using UART. This command capures traces using the serial␣
↪interface.

positional arguments:
  serial_port      Seral port
  serial_baudrate  Seral baudrate
  output_path      Capture output path

options:
  -h, --help       show this help message and exit
```

USB

- Config option - `CONFIG_ZPL_TRACE_BACKEND_USB`
- Command:

```
west zpl-usb-capture [-h] [-t TIMEOUT] [-w] vendor_id product_id output_path

Capture traces using USB. This command capures traces using USB.

positional arguments:
  vendor_id             Vendor ID
  product_id            Product ID
  output_path           Capture output path

options:
  -h, --help            show this help message and exit
  -t, --timeout TIMEOUT
                        Timeout of the USB capture in seconds
  -w, --wait-for-device
                        When this flag is set, the command will wait for the␣
↪device to connect
```

Debugger

- Config option - `CONFIG_ZPL_TRACE_BACKEND_DEBUGGER`
- Command:

```
west zpl-gdb-capture [-h] elf_path output_path

Capture traces using GDB. This command captures traces using GDB from RAM␣
↪using the `dump` command.

positional arguments:
elf_path     Zephyr ELF path
output_path  Capture output path

options:
-h, --help   show this help message and exit
```

Trivial UART in Renode

On top of the above, Renode's simulated trivial UART can be used as well to collect traces in a simulation: `CONFIG_ZPL_TRACE_BACKEND_TRIVIAL_UART`.

2.3.3 Adding named events to traces

Zeppelin provides methods for introducing custom named events to traces from the source code level. To use named events, include the header `zpl/lib.h`, and use the function `sys_trace_named_event()` to generate named events.

2.3.4 Memory profiler

To use Zeppelin memory profiler, y-select the `CONFIG_ZPL_MEMORY_PROFILING` in Kconfig. No further actions are needed in the application code to generate memory profiling events in the generated trace. Memory profiling along with memory events are described in *Memory profiling*.

2.3.5 TLFM events

To use Zeppelin custom events with Tensorflow Lite Micro (TLFM), use the functions `zpl_emit_tflm_begin_event()` and `zpl_emit_tflm_end_event()`, provided by `zpl/tflm_events.h`.

2.4 Testing Zeppelin

To run unit and integration tests, use the following commands:

```
west twister -v -p max78002evkit/max78002/m4 -p max32690fthr/max32690/m4 -p qemu_
↪cortex_m3 -T samples -T tests
```

CHAPTER

THREE

ZEPPELIN CONFIGURATION

The library can be configured at build-time as well as at runtime.

3.1 Configuring the library

To enable Zeppelin tracing support, enable the symbol `CONFIG_ZPL_TRACE` in Kconfig.

3.1.1 Build-time configuration

The build-time configuration can only be selected during build-time. The configuration can either be appended to the conf files:

```
CONFIG_ZPL_MEMORY_USAGE_TRACE=y
```

Or appended to the build command:

```
west build -b stm32f429i_disc1/stm32f429xx samples/trace/memory_profiling -- -
↪DCONFIG_ZPL_MEMORY_USAGE_TRACE=y
```

3.1.2 Runtime configuration

By default, the runtime configuration is turned off with a Kconfig option `CONFIG_ZPL_RUNTIME_CONFIG_NONE`. To turn it on, select one of the following runtime configuration types:

- `CONFIG_ZPL_RUNTIME_CONFIG_UART` - UART shell commands
- `CONFIG_ZPL_RUNTIME_CONFIG_DEBUG` - In-memory debug configuration

UART commands

UART commands runtime configuration can be enabled by selecting `CONFIG_ZPL_RUNTIME_CONFIG_UART` Kconfig option. This option enables runtime configuration via shell module with custom configuration commands.

To display the available configs type `help`:

```
uart:~$ help
Available commands:
  help
  mem_usage_trace
```

Each config can then be either enabled or disabled using the following syntax:

```
<command> enable
<command> disable
```

For example, the memory usage configuration:

```
mem_usage_trace enable
mem_usage_trace disable
```

Debug interface

Debug runtime configuration can be enabled by selecting `CONFIG_ZPL_RUNTIME_CONFIG_DEBUG` Kconfig option. You can use the debug configuration either directly from GDB, or using the `zpl-debug-config` west command. To use it directly in GDB, make sure to load the ELF file with debug symbols. Then set the desired config to `0` (disable) or `1` (enable):

```
set var debug_configs.<config> = <value>
```

For example, to enable the memory usage tracing:

```
set var debug_configs.mem_usage_trace = 1
```

You can also use the `zpl-debug-config` west command:

```
usage: west zpl-debug-config [-h] elf_path config value

Enable/Disable configs in runtime using debug interface. This command can list␣
↪available configs and enable/disable them.

positional arguments:
  elf_path    Zephyr ELF path
  config      Config to set
  value       Value of the config (enable/disable)

options:
  -h, --help  show this help message and exit
```

For example, to enable the memory usage tracing:

```
west zpl-debug-config build/zephyr/zephyr.elf mem_usage_trace enable
```

3.2 Trace formats

The user can then select the following formats:

- Plaintext format, by y-selecting `CONFIG_ZPL_TRACE_FORMAT_PLAINTEXT`
- Common Trace Format (CTF), by y-selecting `CONFIG_ZPL_TRACE_FORMAT_CTF`

3.3 Trace backends

You can choose how the traces will be delivered to the host PC by selecting one of the available tracing backends:

- UART, by y-selecting `CONFIG_ZPL_TRACE_BACKEND_UART`
- USB, by y-selecting `CONFIG_ZPL_TRACE_BACKEND_USB`
- Debugger, by y-selecting `CONFIG_ZPL_TRACE_BACKEND_DEBUGGER`
- Renode's simulated trivial UART, by y-selecting `CONFIG_ZPL_TRACE_BACKEND_TRIVIAL_UART`

Depending on the tracing backend used, the following commands can be used for trace capture.

3.4 Profiling tiers

The library supports three distinct profiling tiers, each offering a different balance of performance impact and tracing detail:

- `ZPL_TRACE_MINIMAL_MODE` - This mode provides extremely lightweight profiling with minimal overhead.
  - It is designed for environments where performance is critical, and persistent trace data is not required.
  - When enabled, it activates basic inference profiling (`ZPL_INFERENCE_PROFILING`) and memory usage tracing (`ZPL_MEMORY_USAGE_TRACE`) to give a high-level view of system behavior without introducing measurable latency.
- `ZPL_TRACE_LAYER_PROFILING_MODE` - This mode enables layer-level timing.
  - It offers more granularity by tracking timing at individual model layers.
- `ZPL_TRACE_FULL_MODE` - This comprehensive profiling mode enables complete tracing.
  - This mode is suitable for in-depth debugging and performance analysis but incurs a higher runtime cost due to its extensive trace capture.
  - It includes all basic and layer-level profiling features, and additionally implies a wide array of kernel and runtime tracing subsystems, which include:
    - syscall tracing,
    - thread scheduling,
    - interrupt service routines (ISRs),
    - synchronization primitives (semaphores, mutexes, condition variables),
    - IPC mechanisms (queues, FIFOs, LIFOs, stacks, message queues, mailboxes, pipes),
    - memory allocators (heap, memory slabs),
    - timers,
    - event handling,
    - polling,
    - power management,

* networking (core, sockets),
* various hardware interfaces (GPIO, idle state tracking).

3.5 Adding new configurations

Depending on the selected Zeppelin integration, the source files in projects need to be adjusted, as described in the following subsections.

3.5.1 Build-time configuration

The build-time configurations can be added to the `zpl/Kconfig` file. The config should follow the Kconfig standard. For example:

```
config ZPL_RUNTIME_CONFIG_NONE
        bool "No runtime configuration"
```

3.5.2 Runtime configuration

New runtime configurations can be defined in the configuration source files. To add a new config, first, using macros, declare two functions in `include/zpl/configuration.h`:

```
ZPL_WAIT_FOR_CONF_DEC(config_name)
ZPL_CHECK_IF_CONF_DEC(config_name)
```

Then, in each configuration type's source file, define those functions, using the same name, and its corresponding Kconfig option. For example in the `configuration_uart.c` file:

```
ZPL_CONF_UART_DEF(config_name, kconfig_option)
```

In the code, there are two ways to use the runtime configuration as guards for the functionalities.

Wait-for function

The "wait-for" function stops the thread, and waits for a signal. Once the signal arrives, the thread is woken up, and continues execution. This is useful when the functionality which this configuration guards is in a separate thread. This wait-for functionality can be called using the macro `ZPL_WAIT_FOR_CONF`. Example:

```
while (true) {
    ZPL_WAIT_FOR_CONF(mem_usage_trace);
    // ... Guarded code
}
```

Check-if function

The "check-if" functionality doesn't stop the thread, and only checks if the configuration is turned on. The function returns a boolean value, which can be checked in a standard "if", to create a guard for the functionality. The check-if function can be called using the macro `ZPL_CHECK_IF_CONF`. Example:

```
if (ZPL_CHECK_IF_CONF(mem_usage_trace)) {
    // ... Guarded code
}
```

CHAPTER

FOUR

MEMORY PROFILING

The memory profiling functionality allows tracing of different memory regions. To use it, enable `CONFIG_ZPL_MEMORY_PROFILING` in the Kconfig.

4.1 Memory types

The profiling tool currently supports the following memory regions in Zephyr:

- Stack
- Heaps
- k_heaps
- Memory Slabs

4.2 Memory event

The memory tracing tool uses custom events. Those events contain information about the memory regions at a point in time.

4.2.1 Common Trace Format

The memory event is a packed structure, described below. It contains information about:

- `timestamp` - the timestamp at which the event occurred
- `id` - id of the event defined as `0xEE`
- `memory_region` - the profiled memory region (stack, heap, etc.)
- `memory_addr` - memory address, which serves as memory region identifier
- `used`, `unused` - Used and unused number of bytes
- `for_thread_id` - ID of the thread associated with the memory region or 0 otherwise

```
typedef struct __packed {
    uint32_t timestamp;
    uint8_t id;
    enum zpl_memory_region memory_region;
    uintptr_t memory_addr;
    uint32_t used;
    uint32_t unused;
```

(continues on next page)

(continued from previous page)

```
    uint32_t for_thread_id;
} zpl_memory_event_t;
```

The available memory regions are defined as an enum:

```
enum zpl_memory_region {
    ZPL_STACK = 0,
    ZPL_HEAP,
    ZPL_K_HEAP,
    ZPL_MEM_SLAB,
};
```

4.2.2 Plaintext

When using a `CONFIG_ZPL_TRACE_FORMAT_PLAINTEXT` format, the memory events are printed in plaintext. The events contain the same information as in the CTF format, but in human readable form. The example shown below, contains a series of memory events of different memory regions.

- `zpl_memory_event` - event name
- `stack`, heap, etc. - memory region
- `0x20011748` - memory address
- `80B 432B` - used and unused memory size (in order)
- `536936848` - thread ID

```
zpl_memory_event stack (0x20011748) 80B 432B 536936848
zpl_memory_event stack (0x20011348) 48B 976B 536936848
zpl_memory_event stack (0x20012148) 48B 272B 536937024
zpl_memory_event stack (0x20012288) 280B 744B 536937200
zpl_memory_event heap (0x20010048) 28B 164B 0
zpl_memory_event heap (0x2001005c) 12B 940B 0
zpl_memory_event heap (0x200106ac) 0B 47132B 0
zpl_memory_event k_heap (0x20010048) 28B 164B 0
zpl_memory_event k_heap (0x2001005c) 12B 940B 0
zpl_memory_event mem_slab (0x2001002c) 256B 7936B 0
```

CHAPTER

FIVE

TRACING CODE SCOPES

Zeppelin allows marking certain code scopes for tracing. When the program execution enters an enabled scope, an event is emitted before and after the code is executed. The scopes can be enabled and disabled at both build-time and runtime.

5.1 Marking the code

5.1.1 Defining the scopes

To use the scopes in your code, first define the scopes using the macro `ZPL_CODE_SCOPE_DEFINE`.

```
// ZPL_CODE_SCOPE_DEFINE(name, is_enabled);
ZPL_CODE_SCOPE_DEFINE(code_scope_name_1, false);
ZPL_CODE_SCOPE_DEFINE(code_scope_name_2, true);
```

The first parameter is the code scope name, which will be used as a handle for marking the code scopes. The second parameter, the flag `is_enabled`, describes the initial state of the scope. The scope can be enabled or disabled at boot, but it can always be switched at runtime.

5.1.2 Marking the code using `ZPL_MARK_CODE_SCOPE`

To mark the desired code scopes, Zeppelin provides a special macro. The scope event will be emitted before and after the code inside is executed. The macro takes in one parameter, which is the name of the scope defined earlier using the macro `ZPL_CODE_SCOPE_DEFINE`.

```
ZPL_MARK_CODE_SCOPE(code_scope_name) {
    // ... Code inside the code scope
}
```

5.1.3 Marking the code using functions

Alternatively, you can mark the scopes using the enter/exit functions. Those functions work exactly the same as the macro showcased above.

```
zpl_code_scope_enter(code_scope_name);
// ... Code inside the code scope
zpl_code_scope_exit(code_scope_name);
```

This method allows for the entry and exit to be defined in separate contexts. For example, you can mark the beginning of a scope in one function, and the ending in a different function.

```
void start_function(void)
{
    zpl_code_scope_enter(code_scope_name);
    // Function body
}

void end_function(void)
{
    // Function body
    zpl_code_scope_exit(code_scope_name);
}
```

5.1.4 Example

In the example below, two scopes are created. Both methods (macro and function) are showcased. Marking the code scopes using both methods results in the same behavior.

```
#include <zpl/lib.h>

ZPL_CODE_SCOPE_DEFINE(code_scope1, false);
ZPL_CODE_SCOPE_DEFINE(code_scope2, true);

int main(void)
{
    zpl_init();

    ZPL_MARK_CODE_SCOPE(code_scope1) {
        // ... Code inside code_scope1
    }

    zpl_code_scope_enter(code_scope2);
    // ... Code inside code_scope2
    zpl_code_scope_exit(code_scope2);

    return 0;
}
```

5.2 Enabling/disabling code scopes at runtime

Code scopes can be enabled and disabled at runtime. This can be done in two ways: via UART shell, or by changing the scope's state in memory.

5.2.1 UART runtime configuration

To use the UART runtime configuration, enable the Kconfig option `CONFIG_ZPL_RUNTIME_CONFIG_UART`.

To list the available configs and their current state, use the command `dynamic_conf list`:

```
uart:~$ dynamic_conf list
Available configs:
 code_scope1: disabled
 code_scope2: disabled
 code_scope3: enabled
 code_scope4: enabled
```

To change the state of the scope use `dynamic_conf enable <scope>` to enable and `dynamic_conf disable <scope>` to disable the scopes:

```
uart:~$ dynamic_conf enable code_scope1
uart:~$ dynamic_conf disable code_scope4
```

5.2.2 Debug runtime configuration

Code scopes can also be enabled directly in the memory, for example by using GDB. If the binary was build with debug symbols, then in GDB you can access them by name.

To check the state of a code scope, in GDB enter:

```
>>> print code_scope1.is_enabled
$1 = false
```

To change the state of a scope:

```
>>> set var code_scope1.is_enabled=1
>>> set var code_scope2.is_enabled=0
```

CHAPTER

SIX

CTF TO TEF CONVERSION

In order to expand the compatibility of the traces received in Common Trace Format (CTF), we created a converter for obtaining them in the Trace Event Format (TEF).

6.1 Usage

You can run the converter with the following command:

```
west zpl-prepare-trace \
  [-h] -o OUTPUT [--zephyr-base ZEPHYR_BASE] [--build-dir BUILD_DIR] \
  [--tflm-model-path TFLM_MODEL_PATH] [--tvm-model-path TVM_MODEL_PATH] \
  ctf_trace
```

There are several optional arguments available:

- `--zephyr-base` - the script tries to automatically find the Zephyr directory in order to use the metadata file definition of CTF events. If this fails, you have to provide the path or set the `ZEPHYR_BASE` environmental variable.
- `--build-dir` - the directory storing the results of the build and ram_report.
- `--tflm-model-path` - the provided TFLite Micro model is processed to extract information about its layer and tensors, and the information is converted to `MODEL` Metadata event.
- `--tvm-model-path` - the provided microTVM graph JSON file is processed to extract information about model's layer and tensors, and the information is converted to `MODEL` Metadata Event,
- `-o` - the file path which points to the file where the converted trace should be saved. If not provided, the JSON will be printed to STDOUT.

Apart from changing the format, this involves a custom logic which can group elements into Duration events, and extend their arguments (see *Converted events*). Completely new events can be added to the trace, extending the context, and improving the trace visualization capabilities (see *Additional events*). Furthermore, the command also converts timestamps from nanoseconds (used in CTF) to microseconds, which are valid for Zeppelin Trace Viewer and Speedscope (it assumes that the timestamps are provided in such unit).

6.2 Events

6.2.1 Converted events

This section describes events that are converted from CTF events, emitted during Zeppelin runtime. Based on those, the trace viewer produces dedicated panels with visualizations (e.g. plot panels).

Zephyr events

For default Zephyr events (defined in metadata), the beginning event always gets the `_enter` suffix, whereas the ending event gets the `_exit` suffix. These events are marked as `B` and `E` events, creating Duration event in TEF. The remaining Zephyr events are of Instant type, but since Speedscope does not support displaying such events, they are also converted to a Duration Event with 1 microsecond of duration.

Example events

```
{
  "name": "isr",
  "cat": "zephyr",
  "ph": "B",
  "ts": 99967.43400000001,
  "pid": 0,
  "tid": 0
},
"...",
{
  "name": "isr",
  "cat": "zephyr",
  "ph": "E",
  "ts": 99980.142,
  "pid": 0,
  "tid": 0
}
```

`zpl_inference`

The Duration event marking the beginning and end of model's inference. It is created from the CTF events `zpl_inference_enter` and `zpl_inference_exit`.

`zpl_inference` events examples

```
{
  "name": "zpl_inference",
  "cat": "zephyr",
  "ph": "B",
  "ts": 0.0,
  "pid": 0,
  "tid": 536912424,
  "args": {
    "thread_id": 536912424
  }
},
"...",
{
  "name": "zpl_inference",
  "cat": "zephyr",
  "ph": "E",
  "ts": 110497.391,
  "pid": 0,
  "tid": 536912424,
  "args": {
    "thread_id": 536912424
  }
}
```

MODEL::{LAYER_OP}[_{SUBGRAPH_IDX}]_{OP_IDX}

The MODEL::* Duration event is unique to each layer of a model, described with:

- LAYER_OP - a tag of operation like CONV_2D or MAX_POOL_2D,
- SUBGRAPH_IDX - an optional number representing the ID of a subgraph to which a given layer belongs,
- OP_IDX - a number representing the ID of the operation in a subgraph.

The event contains:

- identifiers like op_idx and optional subgraph_idx,
- tag with the layer name,
- thread_id pointing to the thread that executed this layer,
- runtime specifying which runtime was used, and
- additional runtime-specific data.

This event is converted from TFLite micro (zpl_tflm_(enter|exit)) and microTVM (zpl_tvm_(enter|exit)) events.

MODEL::* events examples

```
{
  "name": "MODEL::CONV_2D_0_0",
  "cat": "zephyr",
  "ph": "B",
  "ts": 4294973.212,
  "pid": 0,
  "tid": 536912424,
  "args": {
    "thread_id": 536912424,
    "subgraph_idx": 0,
    "op_idx": 0,
    "tag": "CONV_2D",
    "arena_used_bytes": 15408,
    "arena_tail_usage": 88,
    "runtime": "TFLite Micro"
  }
},
{
  "name": "MODEL::CONV_2D_0_0",
  "cat": "zephyr",
  "ph": "E",
  "ts": 4359202.146,
  "pid": 0,
  "tid": 536912424,
  "args": {
    "thread_id": 536912424,
    "subgraph_idx": 0,
    "op_idx": 0,
    "tag": "CONV_2D",
    "arena_used_bytes": 15408,
    "arena_tail_usage": 88,
    "runtime": "TFLite Micro"
  }
}
```

MEMORY

The Metadata event with information about a *memory region* for a given timestamp (`ts`). It is created from the CTF event `zpl_memory`.

MEMORY metadata example

```
{
  "name": "MEMORY",
  "cat": "zephyr",
  "ph": "M",
  "ts": 55.216,
  "pid": 0,
  "tid": 536937200,
  "args": {
    "memory_region": "STACK",
    "memory_addr": 536950376,
    "used": 88,
    "unused": 424,
    "for_thread_id": 536936848
  }
}
```

CPU_LOAD

The Metadata event defining CPU load (cpu_load field) for a given timestamp (ts). It is converted from the CTF event zpl_cpu_load_event.

CPU_LOAD metadata example

```
{
  "name": "CPU_LOAD",
  "cat": "zephyr",
  "ph": "M",
  "ts": 200044.308,
  "pid": 0,
  "tid": 0,
  "args": {
    "cpu_load": 534
  }
}
```

DIE_TEMP

The Metadata event providing DIE temperatures (die_temp array with at most two measurements, in degrees Celsius) for a given timestamp (ts). It is converted from the CTF event zpl_die_temp_event.

DIE_TEMP metadata example

```
{
  "name": "DIE_TEMP",
  "cat": "zephyr",
  "ph": "M",
  "ts": 300327.025,
  "pid": 0,
  "tid": 0,
  "args": {
    "die_temp": [
      21.947092056274414,
      41.94709014892578
    ]
  }
}
```

6.2.2 Additional events

This section contains events that are not produced during the Zeppelin runtime, but are used to provide additional information for better visualizations.

MEMORY::STATICALLY_ASSIGNED_MEM

The Metadata event informing about the part of RAM (in bytes), used by the compiled objects. This value is calculated using size from ram_report and subtracting sizes of the memory regions (from MEMORY *events*). The event looks like this:

MEMORY::STATICALLY_ASSIGNED_MEM metadata example

```
{
  "name": "MEMORY::STATICALLY_ASSIGNED_MEM",
  "cat": "zephyr",
  "ph": "M",
  "pid": 0,
  "tid": 0,
  "ts": 0,
  "args": 14105
}
```

MEMORY::SYMBOLS

The Metadata event contains the mapping of memory region addresses to their symbols extracted from `zephyr.elf`. It is used to present human-readable description of the regions, e.g. making it easier to trace back to the source code. Example event:

MEMORY::SYMBOLS metadata example

```
{
  "name": "MEMORY::SYMBOLS",
  "cat": "zephyr",
  "ph": "M",
  "pid": 0,
  "tid": 0,
  "ts": 0,
  "args": {
    "536950376": "_k_thread_stack_zpl_mem_profiling",
    "536949352": "tracing_thread_stack",
    "536952936": "z_idle_stacks",
    "536953256": "z_main_stack",
    "536938172": "z_malloc_heap",
    "536936540": "_system_heap",
    "536936560": "test_k_heap",
    "536936512": "test_mem_slab"
  }
}
```

thread_name

The Metadata event is used by Speedscope to associate a thread ID with a name. Apart from describing which thread is shown on the flamegraph, the thread name is also used to provide human-readable label for the thread stack.

One event describes exactly one thread with arguments containing the thread name, assigned to the key `name`.

thread_name metadata example

```
{
  "name": "thread_name",
  "cat": "zephyr",
  "ph": "M",
  "pid": 0,
  "tid": 536912424,
  "args": {
    "name": "main"
  }
}
```

MODEL

The `MODEL` Metadata event contains following information:

- `inputs` - the specification of model inputs, described with `name`, `shape` and `dtype`,
- `outputs` - the specification of model outputs, with the same properties as `inputs`,
- `tensors` - the specification of internal data like inputs and outputs of layers, their weights or biases; the properties are the same as `inputs`, but with additional `index` and optional `subgraph_idx`,
- `ops` - the specification of model operations:
  - `op_name` - the name of the operation (e.g. `CONV_2D` or `tvmgen_default_fused_nn_conv2d_add_nn_relu`),
  - `index` - the ID of the operation,
  - `inputs` and `outputs` - lists with indices pointing to `tensors` with input and output data (respectively),
  - `inputs_types` and `outputs_types` - the operation's input and output data types,
  - `inputs_shapes` and `outputs_shapes` - the operation's input and output shapes.

Example event of Magic Wand model for TFLite Micro runtime

```
{
  "name": "MODEL",
  "cat": "zephyr",
  "ph": "M",
  "pid": 0,
  "tid": 0,
  "ts": 0,
  "args": {
    "inputs": [
      {
        "name": "input_1",
        "name_long": "serving_default_input_1:0",
        "shape": [1, 128, 3, 1],
        "shape_signature": [-1, 128, 3, 1],
        "dtype": "float32",
        "quantization": [0.0, 0],
        "quantization_parameters": {
          "scales": [],
          "zero_points": [],
          "quantized_dimension": 0
        }
      }
    ],
    "outputs": [
      {
        "name": "out_layer",
        "name_long": "StatefulPartitionedCall:0",
        "shape": [1, 4],
        "shape_signature": [-1, 4],
        "dtype": "float32",
        "quantization": [0.0, 0],
        "quantization_parameters": {
          "scales": [],
          "zero_points": [],
          "quantized_dimension": 0
        }
      }
    ],
    "tensors": [
      {
        "name": "serving_default_input_1:0",
        "subgraph_idx": 0,
        "index": 0,
        "shape": [1, 128, 3, 1],
        "shape_signature": [-1, 128, 3, 1],
        "dtype": "float32",
        "quantization": [0.0, 0],
        "quantization_parameters": {
          "scales": [],
          "zero_points": [],
          "quantized_dimension": 0
        }
      },
      "..."
    ],
    "ops": [
      {
        "op_name": "CONV_2D",
        "index": 0,
        "inputs": [0, 3, 1],
        "outputs": [10],
        "inputs_types": ["float32", "float32", "float32"],
        "outputs_types": ["float32"],
        "inputs_shapes": {
          "0": [1, 128, 3, 1],
          "3": [8, 4, 3, 1],
          "1": [8]
        },
        "outputs_shapes": {
          "10": [1, 128, 3, 8]
        }
      },
      "..."
    ]
  }
}
```

CHAPTER

SEVEN

ZEPPELIN TRACE VIEWER

Zeppelin Trace Viewer is a visual interface for traces in Trace Event Format (TEF). To fully utilize the features of the viewer, we suggest processing traces with the *CTF converter*.

Go to the GitHub-hosted Zeppelin Trace Viewer to begin analyzing traces.

7.1 Requirements

The viewer leverages workspaces to manage third-party dependencies like Speedscope. This requires yarn in version 1.x or newer, provided by corepack (usually installed together with Node.js):

```
# Allows to use package managers without having to install them
corepack enable
# Downloads specified version of yarn and all dependencies
yarn
```

7.2 Building

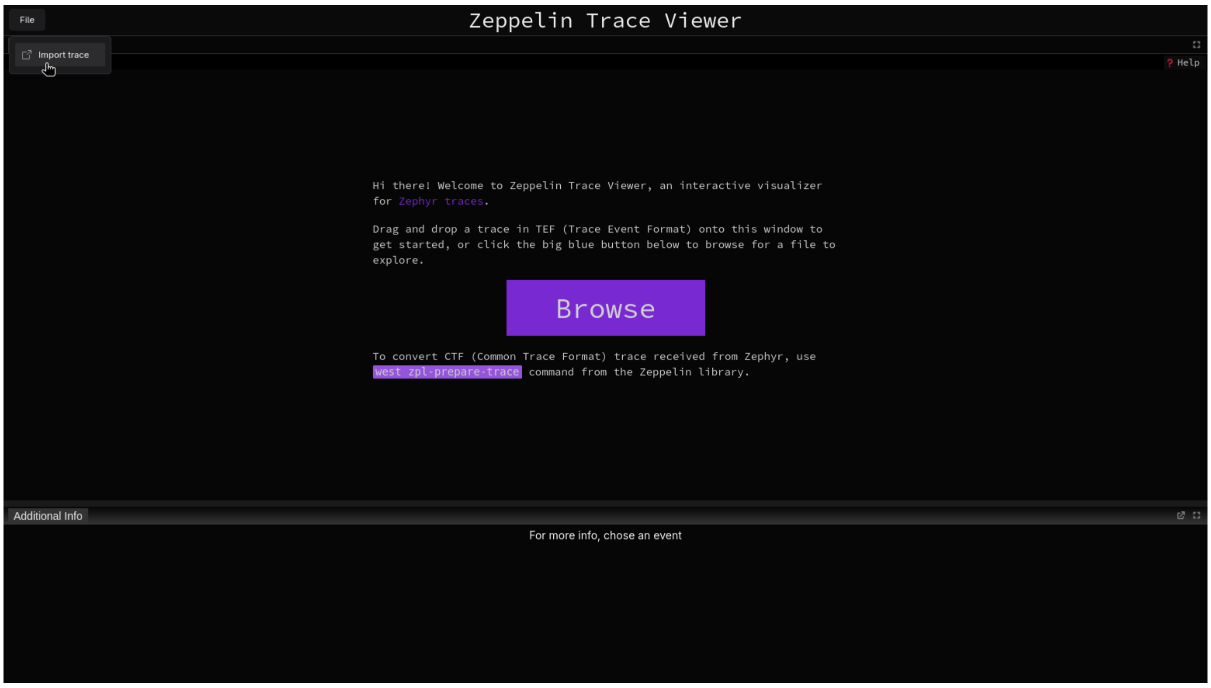
There are several predefined commands with basic utilities:

- `yarn build` - Builds for production, emitting to `dist/`
- `yarn preview` - Starts a server at `http://localhost:4173/` to test production build locally
- `yarn dev` - Starts a dev server at `http://localhost:5173/`
- `yarn lint` - Lints a code and applies all applicable fixes

7.3 Usage

The viewer is built around Speedscope - an interactive flamegraph viewer with support for many trace/profiling formats.

To display a trace, simply import a TEF file using the `File/Import trace` (or `Browse`) button (Fig. 7.1). This action will change the internal state of Speedscope, which triggers the reaction of `Additional info` section. It gathers metadata and initializes custom panels based on available data.



Zeppelin Trace Viewer

Hi there! Welcome to Zeppelin Trace Viewer, an interactive visualizer for Zephyr traces.

Drag and drop a trace in TEF (Trace Event Format) onto this window to get started, or click the big blue button below to browse for a file to explore.

Browse

To convert CTF (Common Trace Format) trace received from Zephyr, use `west zpl-prepare-trace` command from the Zeppelin library.

For more info, chose an event

Fig. 7.1: The main page of Trace viewer

7.3.1 Model info panel

The model info panel displays details about the selected event and the corresponding layer.

Currently, it supports TFLite Micro (Fig. 7.2)

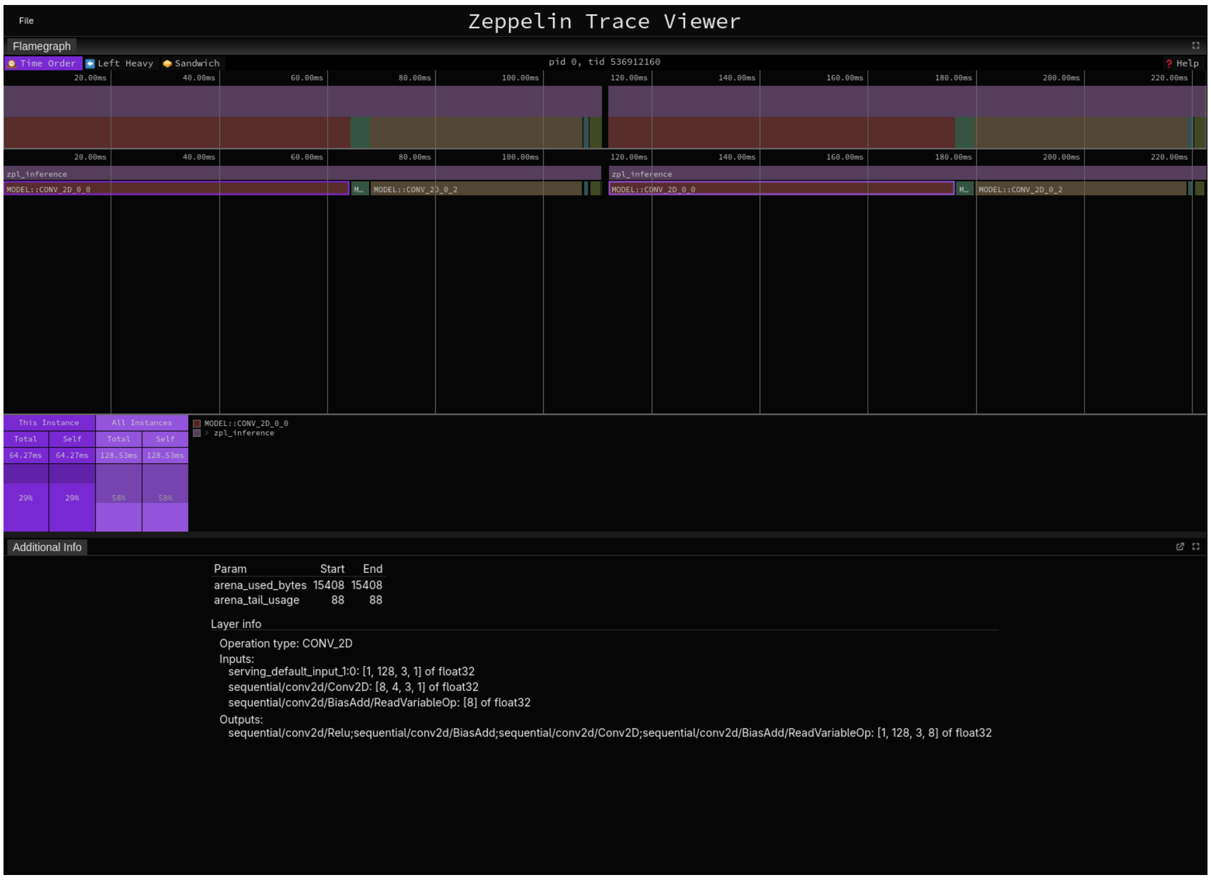
and microTVM runtime (Fig. 7.3).

7.3.2 Plot panels

The trace viewer can also visualize resource usage (like memory or CPU load) or data from sensors (such as DIE temperature) using plots (Fig. 7.4).

Panels with these plots are automatically created whenever a trace with necessary data is loaded:

- *MEMORY* events for memory plots,
- *CPU_LOAD* events for CPU load plot,
- *DIE_TEMP* events for DIE temperature plot.

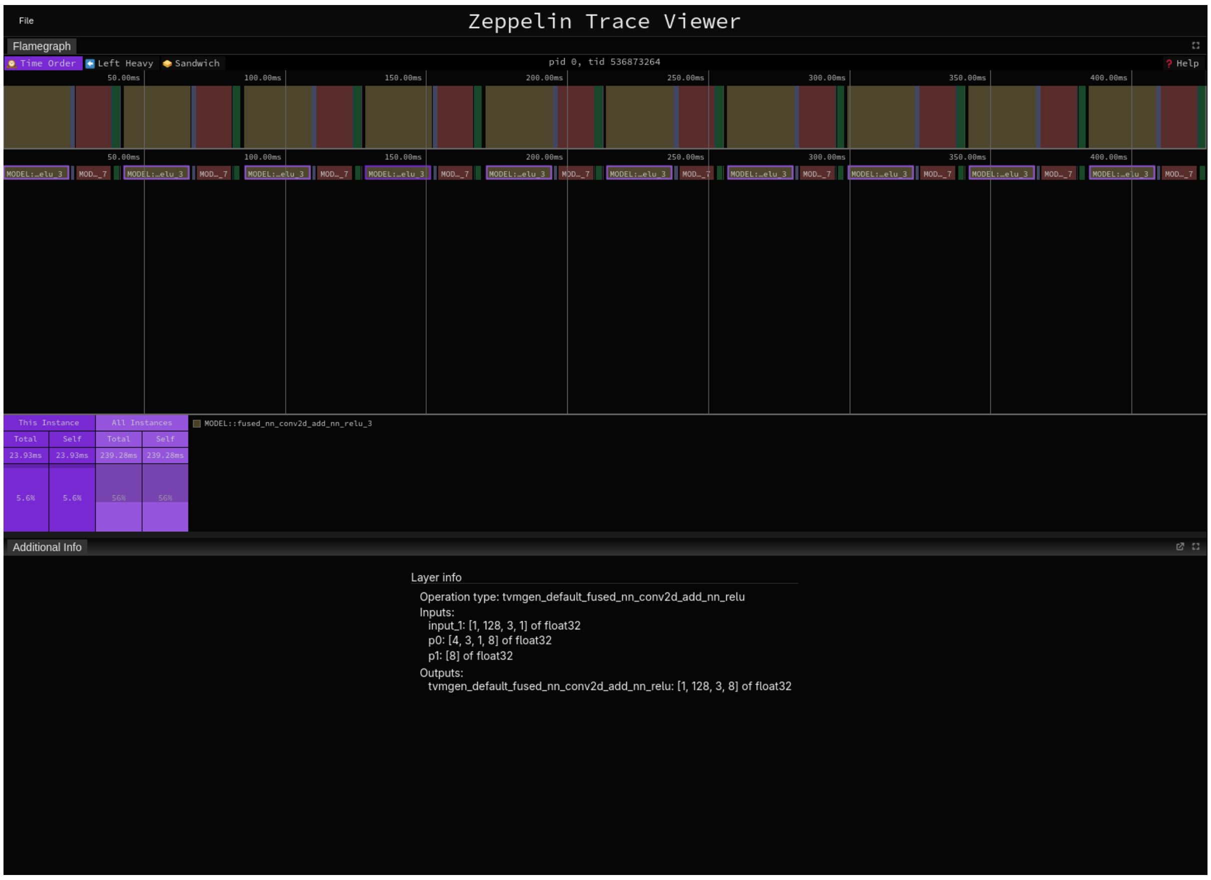


Param	Start	End
arena_used_bytes	15408	15408
arena_tail_usage	88	88

Layer info

Operation type: CONV_2D
Inputs:
serving_default_input_1:0: [1, 128, 3, 1] of float32
sequential/conv2d/Conv2D: [8, 4, 3, 1] of float32
sequential/conv2d/BiasAdd/ReadVariableOp: [8] of float32
Outputs:
sequential/conv2d/Relu;sequential/conv2d/BiasAdd;sequential/conv2d/Conv2D;sequential/conv2d/BiasAdd/ReadVariableOp: [1, 128, 3, 8] of float32

Fig. 7.2: The TFLite micro model information example



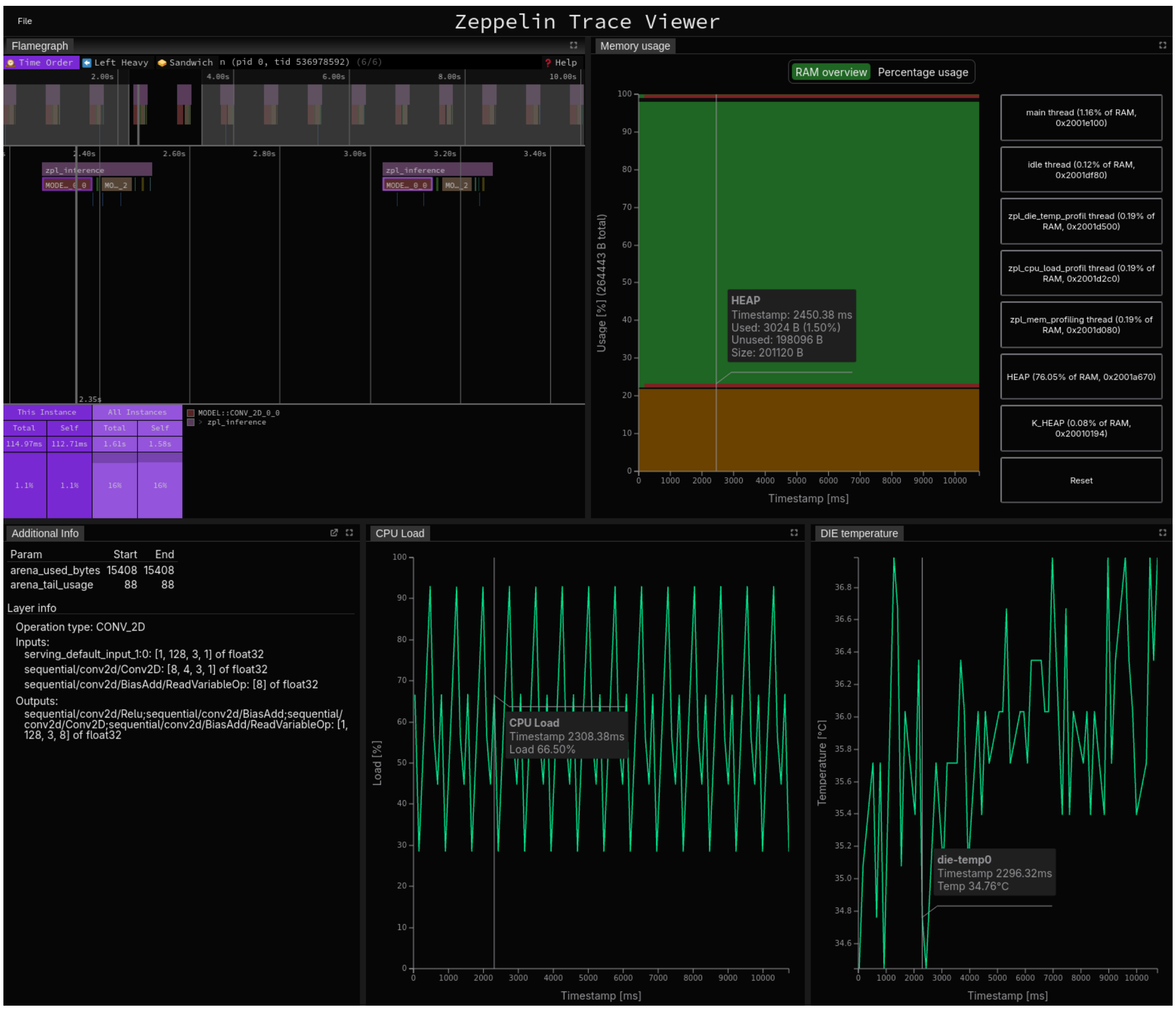
This Instance		All Instances	
Total	Self	Total	Self
23.93ms	23.93ms	239.28ms	239.28ms
5.6%	5.6%	56%	56%

MODEL::fused_nn_conv2d_add_nn_relu_3

Layer info

Operation type: tvmgen_default_fused_nn_conv2d_add_nn_relu
Inputs:
input_1: [1, 128, 3, 1] of float32
p0: [4, 3, 1, 8] of float32
p1: [8] of float32
Outputs:
tvmgen_default_fused_nn_conv2d_add_nn_relu: [1, 128, 3, 8] of float32

Fig. 7.3: The microTVM model information example



Param	Start	End
arena_used_bytes	15408	15408
arena_tail_usage	88	88

Layer info

Operation type: CONV_2D
Inputs:
serving_default_input_1:0: [1, 128, 3, 1] of float32
sequential/conv2d/Conv2D: [8, 4, 3, 1] of float32
sequential/conv2d/BiasAdd/ReadVariableOp: [8] of float32
Outputs:
sequential/conv2d/Relu;sequential/conv2d/BiasAdd;sequential/conv2d/Conv2D;sequential/conv2d/BiasAdd/ReadVariableOp: [1, 128, 3, 8] of float32

Fig. 7.4: The memory usage example