Zephyr® Project

Developer Summit

Leveraging Compiler Code Instrumentation for Tracing & Profiling

Gustavo Romero and
Kevin Townsend, *Linaro*
@gromero @microbuilder

Linaro

Turing: "[...] every time the command goes around the loop it will put a pulse in [the loudspeaker] and you will hear a frequency equal to how long it takes to go around that loop, [...] you would soon learn to listen to that and know whether when [the program] got hung up in a loop or something else or what it was doing all this time [...]". (1950).

Claude Shannon, in "A Mind at Play", p. 108.

- Tracing and Profiling = dynamic analyses (runtime)
- In theory, they can capture almost any info on the system behavior or state as if it was running in real life/production conditions
- However, they can cause an "observer effect", so disturb the runtime. But the disturbances may be negligible, if properly designed and understood

Motivation

- Trace & profile without any financial or license barrier
- Don't depend on additional HW, or proprietary and expensive tooling / software (e.g. J-Link, etc)
- Significant platform limitations (gprof, which only supports a few platforms, often emulated, with no timestamps)
- Show at a very low level and complete way which function in kernel or application code is being called and when
- Have a simple CLI tool for tracing & profiling, like Linux perf and ftrace

Compiler instrumentation

- Several types of code instrumentation are supported by GCC and Clang:
  - code coverage analysis
  - profile-guided optimizations.
    …also used for runtime checks (out-of-bound array check, nullptr, stack)
  - and, finally, for collecting profile statistics!

Compiler instrumentation

Flags of interest here:

1)

```
-finstrument-functions
-finstrument-functions-exclude-file-list=file,file,…
-finstrument-functions-exclude-function-list=sym,sym,…
```

2)

```
-fpatchable-function-entry=N[,M]
-fno-optimize-sibling-calls
```

Compiler instrumentation

```
-finstrument-functions
-finstrument-functions-exclude-file-list=file,file,…
-finstrument-functions-exclude-function-list=sym,sym,…
```

```
void __cyg_profile_func_enter (void *this_fn,
                               void *call_site);
void __cyg_profile_func_exit  (void *this_fn,
                               void *call_site);
```

Instrumentation subsystem for Zephyr

Tracing:

- Ring buffer: 2 working modes (trace buffer and overwriting)
- Event type promotion: based on context

Profiling:

- As functions are called ("discovered") a timestamp is recorded per function and the diff. value is computed at function exit, and accumulated per function
- Heuristics to discard functions and free space for most interesting functions (by name, by number of calls, etc)

- Balancing multiple events vs buffer size:
  - Increase ring buffer size
  - At runtime: Experiment with the trigger and stop addresses to change the trace points and observe different code regions to make the events fit in the ring buffer
  - At compile time: Experiment with the exclude flags to avoid instrumentation by source or function prefixes, removing noise:

```
-finstrument-functions-exclude-file-list=file,file,…
-finstrument-functions-exclude-function-list=sym,sym,…
```

… also per function, using compiler attributes (keyword __attribute__). Instrumentation can be disabled at runtime (internal API), like when entering a critical regions (avoid recursive calls)

- By setting the trace points (setting the trigger and stopper functions), the profile info will be collected between the trace points automatically.

Data Transport

Asynchronous, via serial
Experimenting with other possibilities for synchronous mode:
e.g. SWD (Arm-specific), USB, Ethernet

Stream decoding - Common Trace Format (CTF)

- v1.8 specification (diamon.org/ctf)
- Used on the host side to decode traces from target
- Implemented by libbabeltrace2
- Includes Python3 bindings
- Format details defined in `metadata` file (DSL)
- Some current limitations being address in v2.0 specification (more flexible, JSON objects replacing DSL, etc.)

RFC 57373: Zephyr instrumentation subsystem

Open **Zephyr instrumentation subsystem** #57373
gromero opened this issue on Apr 28 · 3 comments

Introduction

This RFC proposes a new Zephyr subsystem named `instrumentation` meant for profiling and tracing. The new 'instrumentation' subsys leverages the compiler code instrumentation feature available on compilers like GCC and LLVM, enabled via the flag `-finstrument-functions`, to collect profile data, function entry/exit traces, and thread scheduling info in a non-intrusive way.

Problem description

Most profiling and tracing features in Zephyr today either depend on proprietary and expensive tooling, or have significant platform limitations. For instance, Segger's SystemView, which requires J-Link, or gprof, which only supports a few platforms, often emulated, like native_posix, and native_posix_64 platforms, with no timestamp data associated with events. Also, although `tracing` subsys has numerous hooks available in its APIs, these hooks are static and so can't be easily set to trace arbitrary functions in kernel or application code.

By leveraging the compiler code instrumentation feature it is possible to have a platform, tooling, and arch-neutral tracing and profiling subsystem which allows collecting traces at arbitrary code locations, both at compile time and runtime, and at the same time, using the very same mechanism, collect profiling data for the traced code.

Once the traces and profiling data are collected, various transport layers can be used to transfer data from the target to the host, similary to the transport backends available in the `tracing` subsys (UART, USB, etc).

Data can then the viewed in the host using a CLI tool, which can display traces and profing statistics in the text terminal or export it to various formats so data can be visualized on GUI tools, like Perfetto TraceViewer. The same CLI tool allows for setting the functions to be traced at runtime (it can also be set via Kconfig, at compile time).

Proposed change

Implement a new Zephyr subsystem named `instrumentation` and an associated CLI tool that allows setting, collecting, displaying, and exporting tracing and profiling data captured by this new subsystem.

How to enable the Instrumentation subsys

1) Add the following configs to *prj.conf*:

```
CONFIG_INSTRUMENTATION=y
CONFIG_MAIN_STACK_SIZE=4096
CONFIG_RETAINED_MEM_MUTEXES=n
CONFIG_RETENTION_MUTEXES=n
```

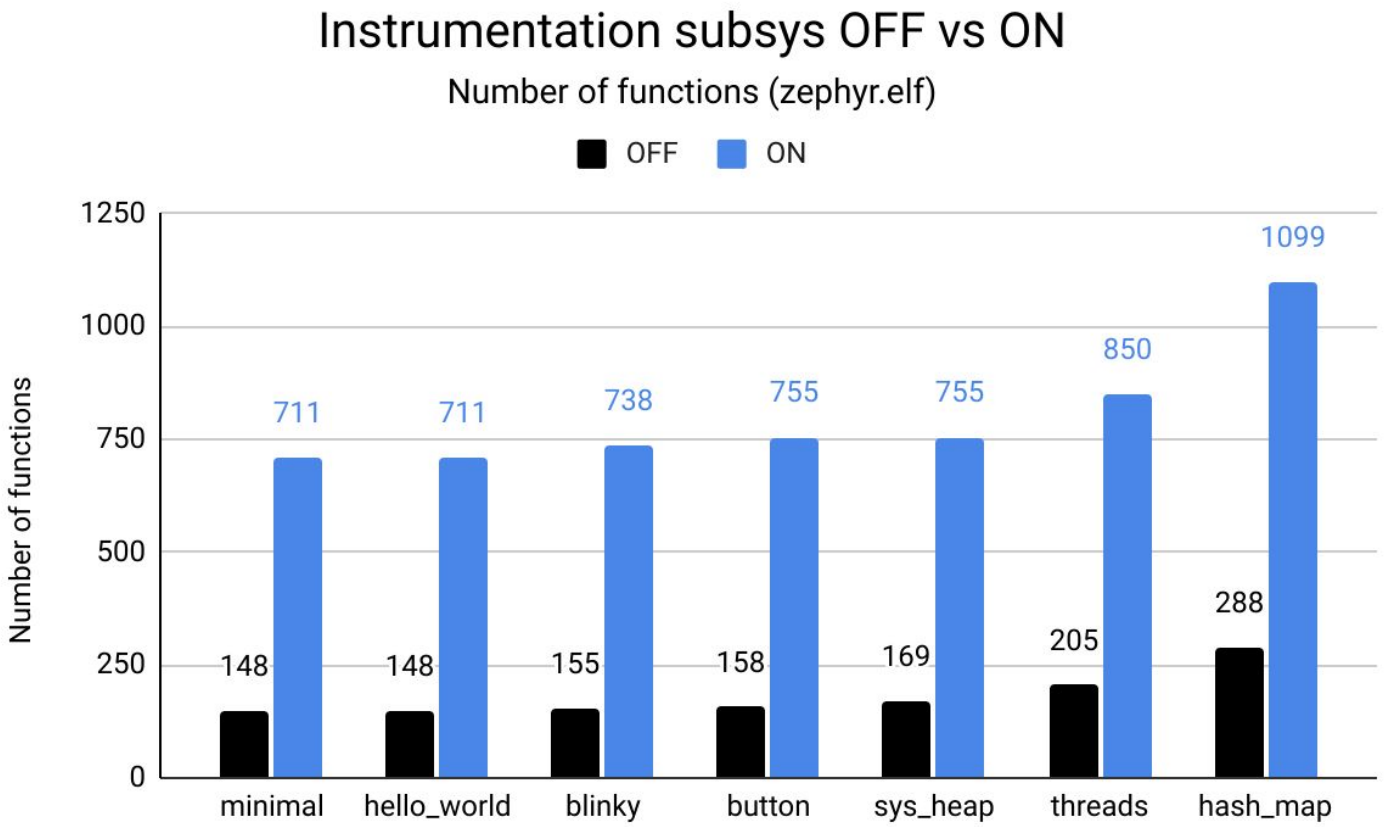
2) Define a new DT node for the retention subsys *boards/<board>.overlay*.

How to turn on the Instrumentation subsys

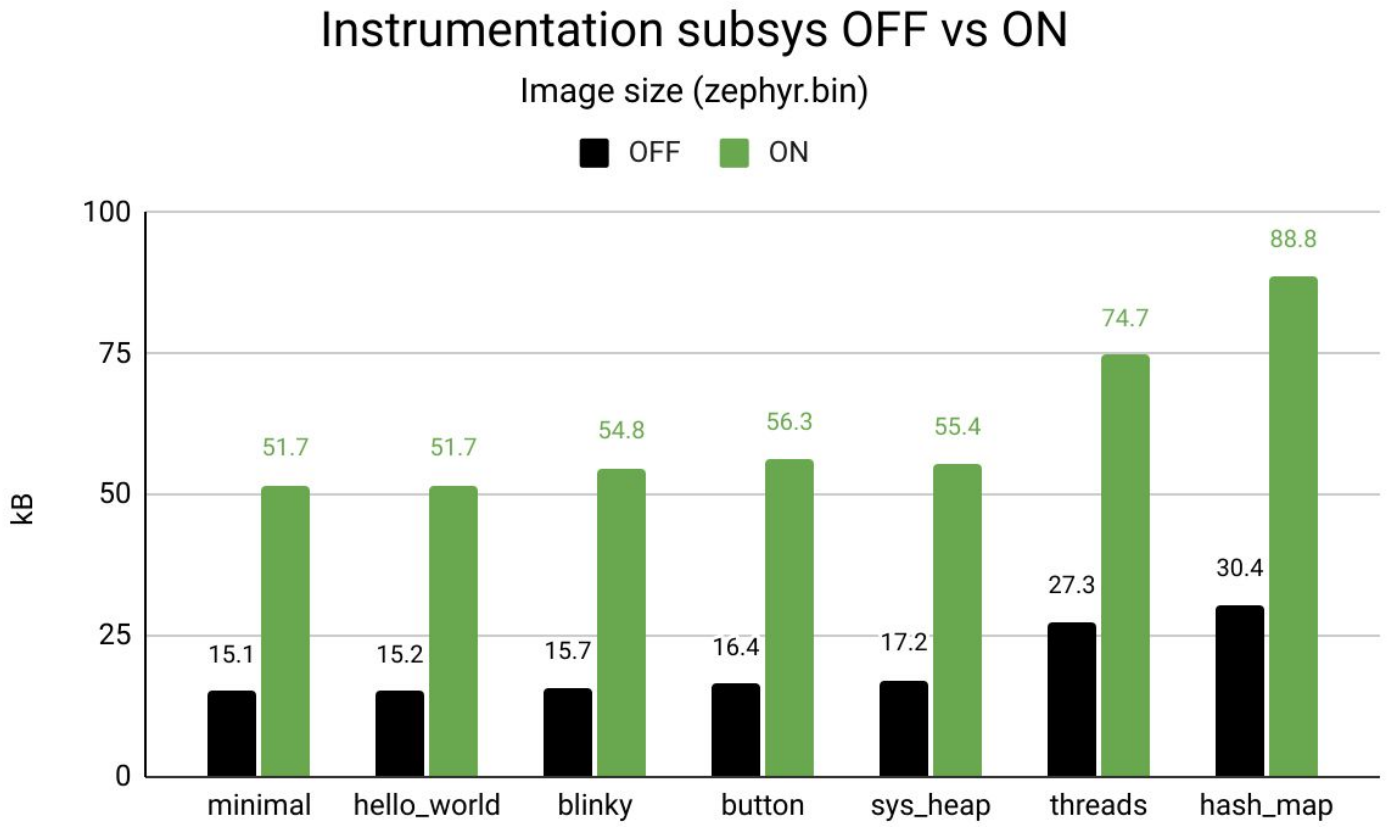
DT node:

```
/ {
        sram@200BF000 {
                compatible = "zephyr,memory-region", "mmio-sram";
                reg = <0x200BF000 0x20>;
                zephyr,memory-region = "RetainedMem";
                status = "okay";

                retainedmem {
                        compatible = "zephyr,retained-ram";
                        status = "okay";
                        #address-cells = <1>;
                        #size-cells = <1>;

                        retention0: retention@0 {
                                compatible = "zephyr,retention";
                                status = "okay";

                                reg = <0x0 0x20>;

                                prefix = [BE EF];
                        };
                };
        };
};

&sram0 {
        reg = <0x20000000 DT_SIZE_K(764)>;
};
```

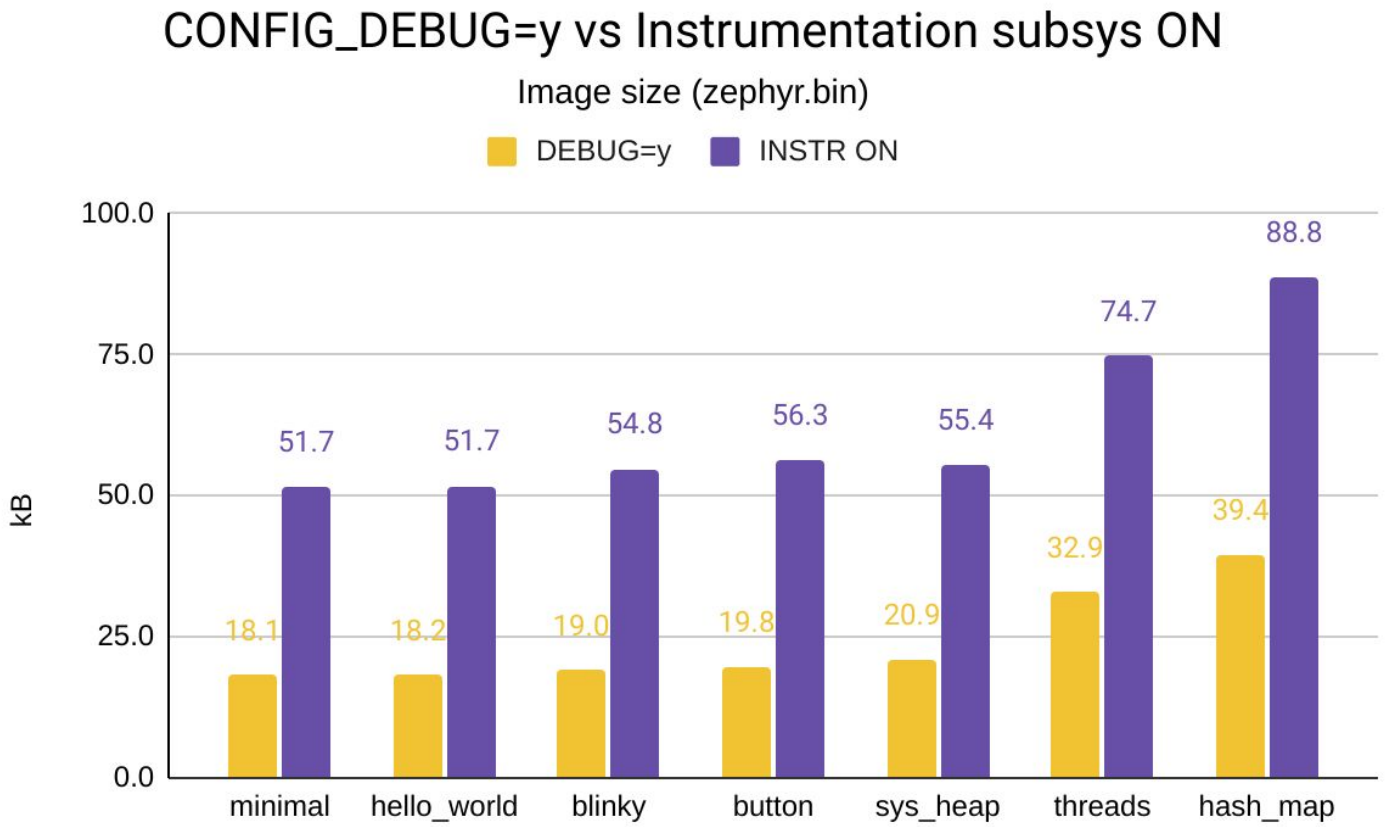
Impact of turning on the Instrumentation subsys



Impact of turning on the Instrumentation subsys

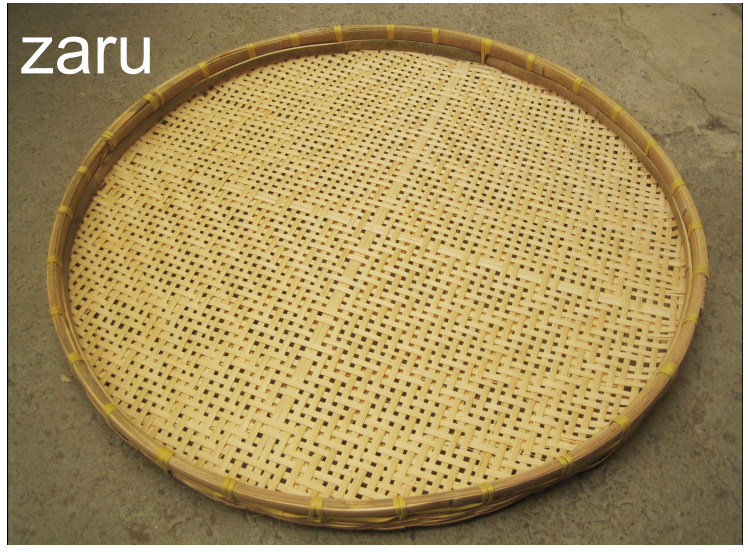


Impact of turning on the Instrumentation subsys



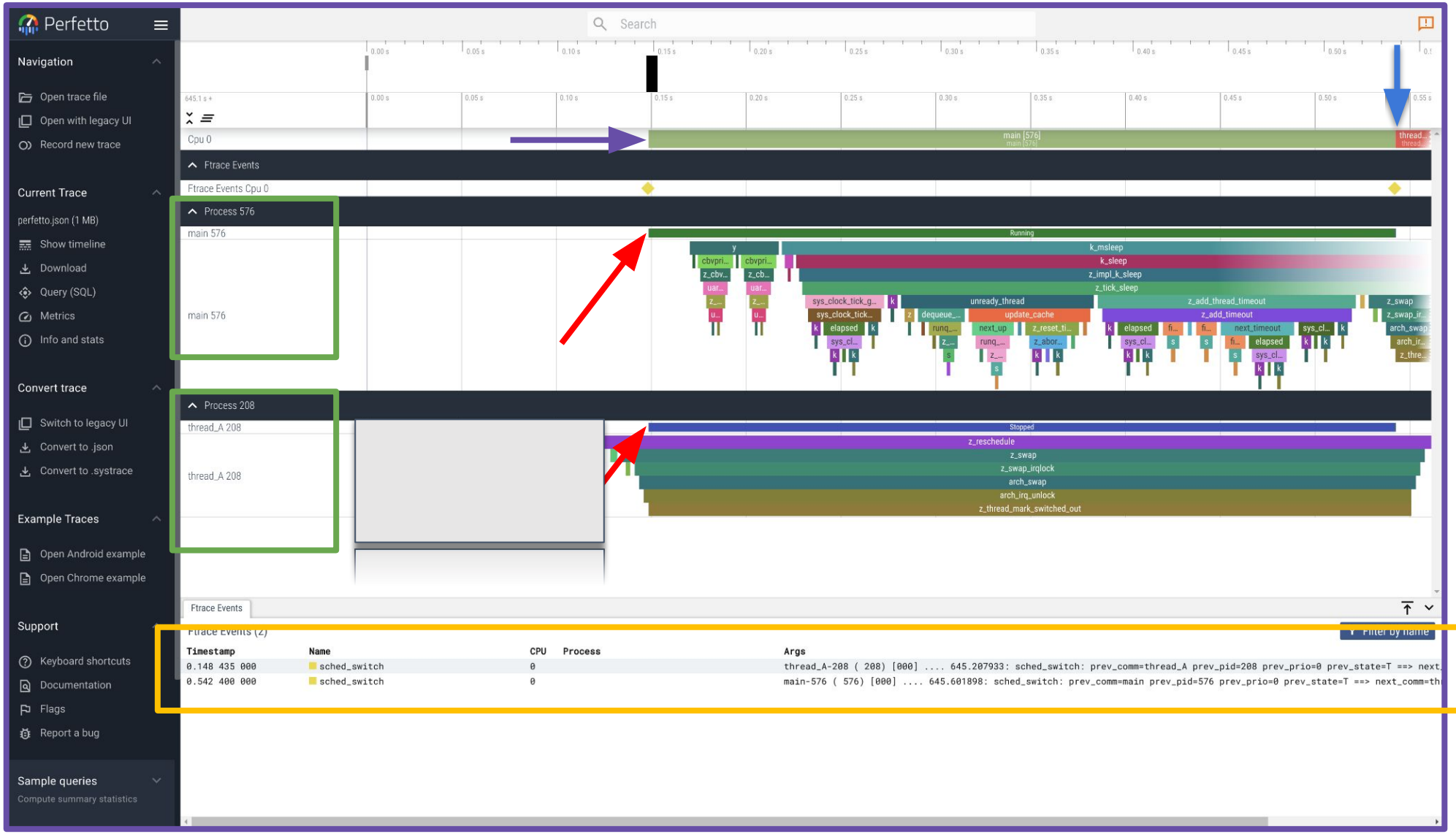
Zaru: A CLI tool for the Zephyr Instrumentation subsys

- Name: zaru ~= strainer, a tool used to filter out or drain water
- Written in Python
- Uses CTF metadata and libbabeltrace2 + Python3 bindings
- Commands: `status`, `reboot`, `trace`, and `profile`
- Example in samples
- Backends:
  - Console / Text
  - Perfetto
  - PDF
  - etc



Data Visualization

Example: Perfetto


Perfetto
Search
Navigation
Open trace file
Open with legacy UI
Record new trace
Current Trace
perfetto.json (1 MB)
Show timeline
Download
Query (SQL)
Metrics
Info and stats
Convert trace
Switch to legacy UI
Convert to .json
Convert to .systrace
Example Traces
Open Android example
Open Chrome example
Support
Keyboard shortcuts
Documentation
Flags
Report a bug
Sample queries
Compute summary statistics
Cpu 0
Ftrace Events
Ftrace Events Cpu 0
Process 576
main 576
Running
k_msleep
k_sleep
z_impl_k_sleep
z_tick_sleep
unready_thread
z_add_thread_timeout
z_add_timeout
dequeue_...
update_cache
elapsed
next_up
next_timeout
Process 208
thread_A 208
Stopped
z_reschedule
z_swap
z_swap_irqlock
arch_swap
arch_irq_unlock
z_thread_mark_switched_out
Ftrace Events
Timestamp
Name
CPU
Process
Args
0.148 435 000
sched_switch
0
thread_A-208 (208) [000] .... 645.207933: sched_switch: prev_comm=thread_A prev_pid=208 prev_prio=0 prev_state=T ==> next
0.542 400 000
sched_switch
0
main-576 (576) [000] .... 645.601898: sched_switch: prev_comm=main prev_pid=576 prev_prio=0 prev_state=T ==> next_comm=th

Compiler instrumentation

```
-fpatchable-function-entry=N[,M]
-fno-optimize-sibling-calls
```

Dynamic Function Instrumentation (DFI)

- WIP/Experimental
- Code is emitted/modified at runtime (like JIT)
- Less overhead at runtime, enabled for specific functions only
- Code generated can be quite flexible, even generated from the host and injected to the target as needed (useful to trace func. arguments?)
- Cons:
  - arch-specific code to implement and maintain
  - tweak the stack (ABI) to manipulate the return address
  - code relocation (to RAM) is necessary, can't patch code in flash
  - still prevents optimizations, like tail call elimination, etc

It's necessary to generate arch-specific code:

```
1. void EmitMOVW_T3_32(uint16_t Rd, uint16_t imm16, void *address)
2. void EmitMOVT_T1_32(uint16_t Rd, uint16_t imm16, void *address)
3. void EmitMOV_T1_16(uint16_t Rd, uint16_t Rm, void *address)
4. void EmitPUSH_T2_16(uint16_t Rt, void *address)
5. void EmitPOP_T3_16(uint16_t Rt, void *address)
6. void EmitBLX_T1_16(uint16_t Rm, void *address)
7.
```

```
/*
 * Patches function at address 'function' with:
 *
 * PUSH R0
 * PUSH R7
 * PUSH LR
 * MOV32 R0, =hook_function
 * BLX R0
 * POP R7 ; POP LR saved value in stack to R7, since we can't pop LR directly.
 * MOV LR, R7
 * POP R7 ; Now pop indeed the saved R7
 * POP R0
 */
int patch_entry(void* function, void* hook_function)
{
        // Due to Arm/Thumb arch interworking shenanigans?
        uint32_t* tmp_f = (uint32_t *)((uint8_t *)function - 1);

        printf("Patching entry of function at %p... \n", tmp_f);

        EmitPUSH_T2_16(R0, tmp_f);
        tmp_f = (uint32_t *)((uint8_t *)tmp_f + 2);
        EmitPUSH_T2_16(R7, tmp_f);
        tmp_f = (uint32_t *)((uint8_t *)tmp_f + 2);

        EmitPUSH_T2_16(LR, tmp_f);
        tmp_f = (uint32_t *)((uint8_t *)tmp_f + 2);

        // LSB
        EmitMOVW_T3_32(R0, (uint16_t)((uint32_t)hook_function & 0xFFFF), tmp_f);
        tmp_f = (uint32_t *)((uint8_t *)tmp_f + 4);
        // MSB
        EmitMOVT_T1_32(R0, (uint16_t)((uint32_t)hook_function >> 16), tmp_f);
        tmp_f = (uint32_t *)((uint8_t *)tmp_f + 4);

        EmitBLX_T1_16(R0, tmp_f);
        tmp_f = (uint32_t *)((uint8_t *)tmp_f + 2);

        EmitPOP_T3_16(R7, tmp_f);
        tmp_f = (uint32_t *)((uint8_t *)tmp_f + 2);
        EmitMOV_T1_16(LR, R7, tmp_f);
        tmp_f = (uint32_t *)((uint8_t *)tmp_f + 2);

        EmitPOP_T3_16(R7, tmp_f);
        tmp_f = (uint32_t *)((uint8_t *)tmp_f + 2);
        EmitPOP_T3_16(R0, tmp_f); // + 2

        return 2 + 2 + 2 + 4 + 4 + 2 + 2 + 2 + 2 + 2;
}
```

```
285 void f(void)
286 {
287     printf("I'm function f\n");
288 }
289
```

```
(gdb) disas f-24
Dump of assembler code for function f:
   0x20000244 <+0>: nop
   0x20000246 <+2>: nop
   0x20000248 <+4>: nop
   0x2000024a <+6>: nop
   0x2000024c <+8>: nop
   0x2000024e <+10>:    nop
   0x20000250 <+12>:    nop
   0x20000252 <+14>:    nop
   0x20000254 <+16>:    nop
   0x20000256 <+18>:    nop
   0x20000258 <+20>:    nop
   0x2000025a <+22>:    nop
   0x2000025c <+24>:    nop
   0x2000025e <+26>:    nop
   0x20000260 <+28>:    nop
   0x20000262 <+30>:    nop
   0x20000264 <+32>:    nop
   0x20000266 <+34>:    nop
   0x20000268 <+36>:    nop
   0x2000026a <+38>:    nop
   0x2000026c <+40>:    nop
   0x2000026e <+42>:    nop
   0x20000270 <+44>:    nop
   0x20000272 <+46>:    nop
   0x20000274 <+0>: push    {r3, lr}
   0x20000276 <+2>: ldr r0, [pc, #8]    ; (0x20000280 <f+12>)
   0x20000278 <+4>: bl  0x20000580 <__printf_veneer>
   0x2000027c <+8>: pop {r3, pc}
   0x2000027e <+10>:    nop
   0x20000280 <+12>:    lsls    r0, r1, #22
   0x20000282 <+14>:    movs    r0, #0
End of assembler dump.
(gdb) n
311     patch_exit(f + patch_exit_offset, profile_func_exit_trampoline);
```

```
(gdb) disas f-24
Dump of assembler code for function f:
   0x20000244 <+0>: push    {r0}
   0x20000246 <+2>: push    {r7}
   0x20000248 <+4>: push    {lr}
   0x2000024a <+6>: movw    r0, #1213   ; 0x4bd
   0x2000024e <+10>:    movt    r0, #8192   ; 0x2000
   0x20000252 <+14>:    blx r0
   0x20000254 <+16>:    pop {r7}
   0x20000256 <+18>:    mov lr, r7
   0x20000258 <+20>:    pop {r7}
   0x2000025a <+22>:    pop {r0}
   0x2000025c <+24>:    nop
   0x2000025e <+26>:    nop
   0x20000260 <+28>:    nop
   0x20000262 <+30>:    nop
   0x20000264 <+32>:    nop
   0x20000266 <+34>:    nop
   0x20000268 <+36>:    nop
   0x2000026a <+38>:    nop
   0x2000026c <+40>:    nop
   0x2000026e <+42>:    nop
   0x20000270 <+44>:    nop
   0x20000272 <+46>:    nop
   0x20000274 <+0>: push    {r3, lr}
   0x20000276 <+2>: ldr r0, [pc, #8]    ; (0x20000280 <f+12>)
   0x20000278 <+4>: bl  0x20000580 <__printf_veneer>
   0x2000027c <+8>: pop {r3, pc}
   0x2000027e <+10>:    nop
   0x20000280 <+12>:    lsls    r0, r1, #22
   0x20000282 <+14>:    movs    r0, #0
End of assembler dump.
(gdb)
```

Questions?
Suggestions?
Comments?

:-)