FrankenTrace: Low-Cost, Cycle-Level, Widely Applicable Program Execution Tracing for ARM Cortex-M SoC

Maciej Matraszek, Mateusz Banaszek, Wojciech Ciszewski, and Konrad Iwanicki
{m.matraszek,m.banaszek,w.ciszewski,iwanicki}@mimuw.edu.pl
Faculty of Mathematics, Informatics and Mechanics
University of Warsaw
Warsaw, Poland

ABSTRACT

Program execution tracing is an important technique in software development and analysis. However, noninvasively obtaining cycle-level traces for modern low-power ARMv7-M-based SoCs is challenging, because convenient off-the-shelf high-speed tracing probes are expensive and cannot be applied to SoCs that lack high-speed debug components, notably Embedded Trace Macrocell (ETM) and parallel tracing port (PTP). To address this issue, in this work, we present FrankenTrace, a technique for generating full, noninvasive, cycle-level program counter traces and full, cycle-level data transfer traces of varying invasiveness on SoCs with only low-speed debug components, namely Debug Watchpoint and Trace unit (DWT), Instrumentation Trace Macrocell (ITM), Single Wire Output (SWO), and an inexpensive probe. We demonstrate the technique by tracing software running on a node of the 1KT testbed.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**; **Traceability**; *Embedded software*.

KEYWORDS

execution trace, memory trace, cycle-level trace, embedded debugging, low-cost tracing probe, ETM, ITM, DWT, ARM Cortex-M

ACM Reference Format:
Maciej Matraszek, Mateusz Banaszek, Wojciech Ciszewski, and Konrad Iwanicki. 2023. FrankenTrace: Low-Cost, Cycle-Level, Widely Applicable Program Execution Tracing for ARM Cortex-M SoC. In *Cyber-Physical Systems and Internet of Things Week 2023 (CPS-IoT Week Workshops '23), May 9–12, 2023, San Antonio, TX, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3576914.3587521

1 INTRODUCTION

Development and analysis of software for modern low-power system-on-chip (SoC) devices, notably those based on ARM Cortex-M processors, pose particular challenges due to the limited resources such devices provide. An important technique in this process is *tracing*: an ability to obtain a detailed record of how software is executed by an SoC's processor (CPU). Specific trace types are required for different purposes. In particular, an instruction-level *PC trace* records subsequent values of the program counter (PC) register, which holds the address of the instruction currently executed by the CPU. Combined with the software binary, a PC trace gives a full record of subsequently executed instructions. On the other hand, to examine data transfers issued by the CPU to memory and memory-mapped interfaces of the SoC's various components, an *LSU trace* is needed. It records addresses, values, and directions (read/write) of data transfers performed by the so-called Load Store Unit (LSU), which manages the transfers in ARM processors.

Instruction-level traces have many applications, not only in software development and debugging, but also in research reproducibility, post-mortem experiment analysis, or emulators. However, analyzing timing issues and precise benchmarking require more detailed *cycle-level traces*, which record the executed instruction or data transfer for each clock cycle of the processor. Cycle-level traces unveil, for instance, which processor instructions require more than one cycle to be executed, and how long it takes to complete each data transfer. Moreover, to ensure a proper interpretation of the results, many applications necessitate *noninvasive tracing*, that is, one that does not alter the execution process itself.

Arguably, the most convenient way to obtain execution traces for a piece of software for a given device is to use an emulator of the device. During emulation, executed instructions and data transfers can be traced in a noninvasive way by monitoring the operation of the emulator. However, functional emulation tools capable of simulating the ARMv7-M architecture (i.e., ARM Cortex-M3, Cortex-M4, and Cortex-M7 processors), such as QEMU, Renode, or ARM Fast Models can be used to generate only instruction-level traces, as they do not simulate the hardware in sufficient detail to obtain cycle-level traces. Cycle-level tracing would require precise simulation of the entire SoC, including the CPU, memory modules, and the SoC's other components. Even a minor difference between simulation and execution on actual hardware may result in a significantly distinct program flow, making it likely that some issues might be observed only with real hardware. While in the case of the CPU there exist cycle-exact models compiled from hardware's sources (i.e., RTL), such as Arm Cycle Models, high-fidelity models for complete SoCs are hardly available. Consequently, today, cycle-level tracing in principle must be performed on real hardware.

Designers of modern processors recognize this need for hardware-assisted tracing. In particular, the ARMv7-M architecture [4] specifies multiple optional debug components, which provide various levels of insight into processor operation. Typical high-fidelity noninvasive tracing solutions utilize off-the-shelf high-speed commercial tracing probes, such as Segger J-Trace, Keil ULINKpro, IAR I-jet Trace, and accompanying software. However, such probes can

Publication rights licensed to ACM. ACM acknowledges that this contribution was authored or co-authored by an employee, contractor or affiliate of a national government. As such, the Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only.
CPS-IoT Week Workshops '23, May 9–12, 2023, San Antonio, TX, USA
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0049-1/23/05…$15.00
https://doi.org/10.1145/3576914.3587521

Table 1: Low- and high-speed debug components featured in representative (as of Feb. 2023) Cortex-M3-based SoCs.

SoC series	low-	high-speed	SoC series	low-	high-speed
CC26x0	✓	✗	ADUCM360/3029	✗	✗
SAM3	✓	✗	PSoC 5LP/FM3	✓	some
STM32F100	✓	✗	STM32F103	✓	some
EFM32G/TG	✓	✗	EFM32LG/GG	✓	✓
LPC1300	✓	✗	LPC1800	✓	✓

be even a few orders of magnitude more expensive than a traced SoC. Therefore, their availability to software engineers is limited, and it is virtually impossible to deploy them within large-scale IoT testbeds such as 1KT [6] or IoT-LAB [2]. Furthermore, many commercial SoCs do not support high-speed probes, as they do not have the necessary high-speed debugging components: Embedded Trace Macrocell (ETM) and parallel trace port (abbr. PTP in this paper). On the other hand, low-speed debug components—Debug Watchpoint and Trace unit (DWT), Instrumentation Trace Macrocell (ITM), and Single Wire Output (SWO)—are featured in most modern SoCs with ARMv7-M CPU (see Table 1). For instance, Cortex-M3 SoCs deployed in the 1KT, IoT-LAB, and Indriya2 [3] testbeds feature all low-speed debug components but no high-speed ones.[1] As a result, more widespread and cost-effective tracing solutions are desirable.

In this paper, we present FrankenTrace, a technique for generating full and accurate traces by means of only low-speed debug components (DWT, ITM, and SWO) and without any expensive tracing probes but just significantly cheaper logic analyzers or UART-USB chips (e.g., FT232RL). FrankenTrace supports generating two types of traces: a noninvasive cycle-level PC trace and a cycle-level LSU trace of varying invasiveness (with a trade-off between invasiveness, trace completeness, and tracing time). When combined, the traces show which instruction is executed in each processor cycle, and what value is read from or written to which address in that cycle. We demonstrate FrankenTrace in action by tracing software running on a CherryMote, a node of the 1KT testbed, and argue that the method can be deployed at a low cost at a large scale.

The rest of this paper is organized as follows. First, we discuss related solutions and ARMv7-M debug components (Section 2). Then, we present the idea behind FrankenTrace (Section 3), details of PC tracing (Section 4), LSU tracing (Section 5), and the hardware required to capture traces (Section 6). Finally, we present a case study (Section 7) and conclude outlining future work (Section 8).

2 BACKGROUND

A naïve low-cost approach to generating a trace for a piece of software executed on an SoC is to use a JTAG-based debugger and step through subsequent instructions. This approach is not viable for tracing long executions, though, as, for example, 100 ms of execution on a 48-MHz CPU requires nearly 5 million debugger steps. Moreover, halting the CPU after each instruction is highly invasive. A noninvasive trace could be obtained by modifying the previous approach: stop the execution after the first instruction, restart the execution and stop after the second instruction, and so on, executing one more instruction each time before halting the CPU. However, this method is too inefficient for long executions.

As an alternative to off-the-shelf tracing probes, a community-driven project Orbcode [1] develops an open-source FPGA-based probe and software tools for debugging and tracing Cortex-M-based SoCs. However, the probe does not support communication speeds high enough to noninvasively trace an execution.

HATBED [11] demonstrates how low-cost hardware can enable profiling in IoT testbeds. However, to capture every executed instruction that technique requires ETM, which invasively halts the CPU when its output buffer overflows. We build upon the idea and show how this type of hardware can be exploited to obtain a full noninvasive cycle-level PC trace and a full cycle-level LSU trace even from an SoC that does not feature ETM.

2.1 ARMv7-M Debug Components

To facilitate tracing software executed by a CPU, the ARMv7-M architecture specifies multiple debug components. Although they are optional, virtually all modern SoCs based on Cortex-M3 or M4 contain a subset at minimum.

DWT: The Debug Watchpoint and Trace unit provides multiple performance counters, for instance, a counter that is incremented on each cycle of the processor clock. Moreover, DWT features comparators that can be set up to compare a configured value with addresses or values of data requests issued by the processor, addresses of instructions, or values of the cycle counter. Upon a match, DWT can generate a packet (subsequently passed to ITM) detailing the matched data or instruction, generate a debug exception, cause the processor to halt and enter a debug state, or signal the match to another component. DWT can be also configured to periodically sample PC and pass its current value to ITM.

ITM: The Instrumentation Trace Macrocell captures tracing packets generated by DWT. It also allows for emitting events in software by writing to its registers, thereby supporting `printf`-style debugging. ITM may optionally timestamp each received event by appending a special packet to the output. If the received packets overflow its output buffer, ITM arbitrates which are discarded and notifies about the overflow. The packets are then delegated to TPIU.

ETM: One of the features of the Embedded Trace Macrocell is the ability to generate a real-time PC trace. Although ETM employs buffering and heavy compression of the produced information, a significant amount of bandwidth is required to output all tracing packets. Nevertheless, ETM can be configured to halt the processor when the output buffer overflows. Commercial off-the-shelf tracing probes use ETM for execution tracing. However, in contrast to DWT and ITM, which are present in virtually all popular ARMv7-M SoCs, the high-speed ETM is not available in many of them (see Table 1).

TPIU: The Trace Port Interface Unit processes tracing packets generated by ITM and ETM, and handles outputting them to an external analyzer for further processing. TPIU can output data via two interfaces:

- asynchronous Single Wire Output (SWO)—using NRZ encoding (i.e., standard 8N1 UART) or Manchester encoding (improved signal stability at higher communication speeds),
- parallel port (PTP)—consisting of a clock line and multiple data lines (Cortex-M-based SoCs feature up to 4 data lines).

[1]Texas Instruments CC2650 SoC featured in 1KT and Indriya2 has no ETM and PTP. STM32F103REY SoC featured in IoT-LAB's M3 and A8-M3 boards has no PTP pins.

Assuming the processor is clocked at n MHz, a 4-line PTP may output up to $8n$ Mbps [5]. SWO outputs up to $\sim 0.72n$ Mbps using NRZ encoding [5], but at higher baud rates the Manchester encoding is required to reliably decode the signal, resulting in up to $\sim 0.48n$ Mbps [5]. However, in practice, cheap off-the-shelf components can reliably receive UART transmission with a symbol rate of only up to 8–12 Mbaud (~6.4 Mbps) [7, 9]. Using SWO with a USB logic analyzer and software decoding, we are able to achieve communication clocked at 8 MHz—the same as the tracing output supported by the hardware in HATBED [11]. While popular SoCs widely support SWO, not all of them feature PTP that is required for real-time tracing with commercial off-the-shelf probes (see Table 1).

3 PRINCIPAL IDEA BEHIND FRANKENTRACE

As a point of reference, let us consider the Texas Instruments CC2650 SoC [10], deployed as an experimental node in the 1KT testbed. This SoC is clocked at 48 MHz and features two single-cycle memories. A tracing packet with a value of PC, as sampled by DWT, has a size of 5 B. A timestamp packet of ITM, in turn, has a size of 1–5 B. With the 6.4 Mbps of SWO throughput, tracing is inherently limited to about 100K samples per second. In effect, PC can be sampled approximately only once every 512 CPU cycles.

In contrast, if TI CC2650 had ETM and a 4-data-lane PTP, it could output traces at up to 384 Mbps. Such a bandwidth would be sufficient in almost all cases. However, imagine an extreme case: a loop composed of indirect jumps between the SoC's two single-cycle memories. Each such jump lasts 3 CPU cycles and generates an ETM tracing packet of 4 B. Therefore, a bandwidth of 512 Mbps would be required to output complete traces generated by this loop without ETM stalling the CPU.

These estimates highlight the core idea behind FrankenTrace: even with high-speed hardware, in extreme cases, outputting a full trace can exceed the available bandwidth. When lower-speed, but more widely available hardware is utilized, this bandwidth issue becomes the key problem to overcome. FrankenTrace copes with it by executing the same piece of software multiple times and, if necessary, inserting delays. Tracing results are then stitched together to recreate a full cycle-level PC and LSU trace. See Figure 1 for an overview of FrankenTrace's operation.

4 PC TRACE

One feature of FrankenTrace is the ability to generate accurate cycle-level PC traces using low-speed debugging hardware. In particular, FrankenTrace guarantees a lack of gaps in the trace and the absence of alterations due to artificial CPU stalls during tracing. This is an advantage over the stall-based ETM tracing, which is not a reliable source of instruction timings. For example, when trying to capture a tight loop involving data transfers to an SoC's component, stalling would obscure the influence of the response time of that component. To some extent, this has the same shortcomings as clocking the CPU at a lower frequency when tracing. In contrast, FrankenTrace's goal is to generate a faithful trace as if the execution proceeded without any tracing. As a consequence, the developed method should be noninvasive, or at least be able to reliably reconstruct true events after an introduction of invasiveness.

Table 2: DWT and ITM configurations for various output baudrates from a 48 MHz-clocked TI CC2650 that optimally use the available bandwidth. TPIU is configured to use SWO with UART 8N1 encoding and bypass the packets formatter.

baud rate	N	TPIU prescaler	DWT_CTRL[12:0]
1 Mbaud	5120	47	0x1209
2 Mbaud	3072	11	0x1205
8 Mbaud	512	5	0x100f
24 Mbaud	192	1	0x1005
48 Mbaud	128	0	0x1003

4.1 Overcoming the Throughput Limit

Since we cannot directly collect a full PC trace in a noninvasive way, we configure DWT to sample PC at exactly N-cycle intervals, where the value of N is configurable (see Table 2). For N sufficiently large, the packet generation rate does not exceed the available bandwidth, allowing us to faithfully capture a partial trace. We execute the traced software N times, capturing partial traces at all possible cycle offsets. More precisely, the i-th repeated execution yields PC values with cycle counts $t = k \cdot N + i$, for $k \in \mathbb{N}$. The partial traces are then combined into a full PC trace.

4.2 Ensuring Repeatability

FrankenTrace relies heavily on repeated execution, requiring each run of traced software to be identical, except for purposeful alterations from the tracing mechanism. This requirement necessitates some consideration. For instance, inherently nondeterministic SoC's components are a possible source of inconsistency. In general, analog components, especially asynchronous clocks, are major sources of nondeterminism. However, in Cortex-M-based SoCs, it is often possible to mitigate the problem with clocks by using dividers to derive all clock signals from a single source. In other cases, where the execution is dependent on timing that cannot be controlled (e.g., on an input from an SoC's environment sensor), it is necessary to add a resynchronization routine to the traced code. The routine restores the PC sampling cycle offset (denoted by i above) and designates a point of reference for combining partial traces.

A naïve approach to repeated execution is to invoke a block of the program within a for-loop. This idea, however, has a lot of inherent difficulties, as the state of various caches and other SoC's components may differ between the iterations, and preventing this is a tedious task. The solution to this problem, which ensures a consistent state before all executions, is to reset the device before each repetition. ARMv7-M defines two reset levels: *local* and *power-on*, and provides two standard reset mechanisms through the *Application Interrupt and Reset Control Register* (`AIRCR`):

- The `VECTRESET` control bit causes a local reset as a debug feature, often referred to as a *soft reset* or *CPU reset*. This typically resets only CPU, although the architecture allows it to reset other SoC's components as well.
- The `SYSRESETREQ` control bit requests a reset by a system external to CPU. This is also known as a *system reset*, which typically resets the entire SoC.

If `SYSRESETREQ` does not trigger a full chip reset, the vendor usually provides another mechanism for such a reset (e.g., a special register).

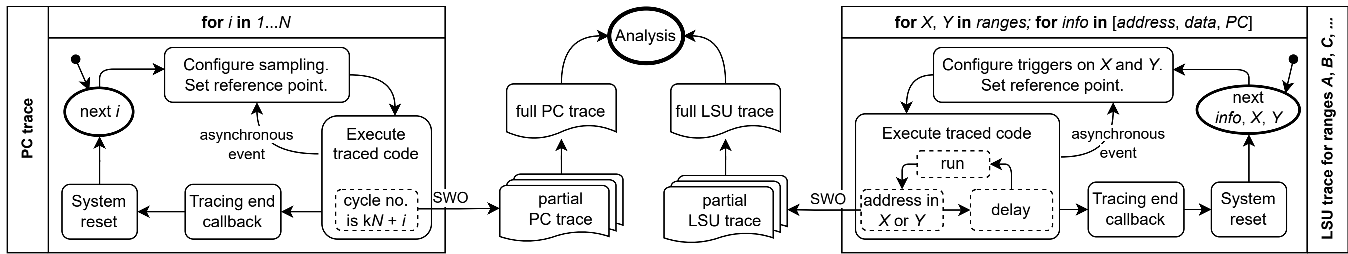

Figure 1: Overview of FrankenTrace's operation.

A Cortex-M-based SoC typically provides a dedicated hardware register to distinguish between a power-on reset (power-cycle reset), a reset-pin reset, a CPU reset, and a system reset. In our experiments with the TI CC2650, a system reset appears to act like a pin reset, that is, most of the components transition into their reset states. Therefore, to end a tracing iteration, the traced software calls FrankenTrace's hook function that triggers a system reset. On TI CC2650 it takes ∼1 ms to reset and proceed to the next iteration.

We have experimentally verified that DWT and ITM can be configured to enable repeatable tracing at any point of software. For instance, we confirmed that traces that started at the beginning of a reset handling routine yielded consistent results. Consequently, the code that enables tracing can be used as a resynchronization point after the loss of deterministic execution. Moreover, we verified that the trace is properly generated even when the traced software puts the SoC into sleep (`idle` power mode) and then wakes it up.

4.3 Varying the Trigger Point

Since the system is fully reset between each repetition of an execution, a procedure is necessary to determine what sampling offset, *i*, is required during the current iteration. In other words, a value needs to be reliably retained between system resets. A standard approach, storing the value in the flash memory, could strain this component's lifetime, because typical flash memories in Cortex-M-based SoCs are rated for up to 10^5 erase/write cycle [10].[2]

Instead, on the platform we use for evaluation, one can rely on SRAM retention during a reset. In fact, a typical SRAM cell has no reset line and loses its state only upon a loss of power. With TI CC2650, as long as the traced code is not powering off SRAM, the tracing metadata would be preserved throughout the entire procedure. However, it is important to ensure that the values are not overwritten by any boot code (e.g., by placing them in an uninitialized program memory section).

Finally, we ensure that the sampling offset does not affect the behavior of the ITM/DWT configuration code, except for the values written to the registers of the debug components. This way changing the sampling offset does not change the execution of the traced code. Since PC is sampled upon a transition of a configured tap bit of the cycle counter, we can influence the point at which the sample is taken by carefully manipulating the DWT configuration and by modifying the initial value of the counter.

[2]Flash memory is mainly worn out by the erase operation, so a scheme using only one-way bit-by-bit writing could possibly be devised to partly mitigate that problem.

4.4 Reconstructing a Trace

Traces collected from all execution repetitions are combined to form a full PC trace. Several features of FrankenTrace facilitate this procedure and help verify the correctness of the resulting trace.

At the beginning and end of each repetition, the tracing code emits packets with the iteration metadata via ITM registers. This simplifies splitting the packet stream output by the whole tracing procedure into separate traces for each iteration. Moreover, before the device is reset at the end of a repetition, the tracing code disables ITM event sources and waits for ITM to flush all its output buffers. In particular, this approach makes it easier to properly capture debug packets with a logic analyzer or a UART-USB dongle.

While a full system reset resolves most of the repeatability issues, even the tracing code responsible for configuring DWT and ITM and emitting metadata at the beginning of each iteration can introduce nondeterminism. FrankenTrace takes multiple measures to ensure the same state of the processor and memory subsystem after each execution of this routine. These include executing instruction and memory barriers, flushing memory bus buffers, and synchronizing clock frequency domains (which is possible, for example, by exploiting side effects of sync-down bridges during data transfers).

To validate the correctness of constructed traces, FrankenTrace always executes a special piece of code with a known trace before the analyzed code, so that later the trace can be validated by verifying the fragment corresponding to the special code. Moreover, some execution iterations are done redundantly to assess the consistency of the output. Likewise, substantially redundant timestamp packets generated by ITM after each PC sample enable identifying any corruption and facilitate reconstructing the full trace.

Finally, the tracing setup code generates an initial ITM packet. Since ITM timestamps are relative to previous ones, this gives a point of reference for timestamps of packets in the actual trace.

5 LSU TRACE

The goal of an LSU trace is to record for each data transfer issued by CPU the transferred data, its address, its direction (read/write), and the current PC value. To this end, we employ DWT comparators, which can be configured to emit this information when an address of an ongoing transfer matches a comparator's observed address range. However, only four comparators can be used for this, and a single address range is expressed by specifying 16–32 most significant bits of an address. As a consequence, transfers within only four 64 KiB long ranges can be traced at once. Moreover, sizes of the involved tracing packets range from 2 B to 5 B, so information emitted by

frequent transfers (e.g., in a busy-waiting loop) might exceed the available output bandwidth.

5.1 Overcoming the Range Limit

FrankenTrace's LSU tracing employs an approach similar to its PC tracing: the traced software is executed multiple times, and each time transfers within another address range are traced. In practice, only a fraction of the entire 32-bit address space is actually used, as SoCs usually feature only a limited amount of memory, a dozen components with memory-mapped registers, and a handful of special-purpose regions (e.g., bit-banding or nonbufferable aliases). This greatly limits the number of 64 KiB ranges that need to be traced. Furthermore, the total tracing time can be reduced by recording only interesting ranges. For example, to examine interactions between CPU and a low-power wireless radio, it is enough to trace only transfers within an address range of the radio registers.

5.2 Overcoming the Throughput Limit

There is no configuration option to trigger DWT only exactly every N-th address match. Therefore, we cannot employ a solution similar to PC tracing such as capturing every N-th data transfer and repeating the procedure multiple times with increasing offsets.

There are two ways to decrease the packet generation rate noninvasively. The first one is to execute the traced software three times and record the different packet types (address, data and direction, PC value) separately in each run. The second is to decrease the size of traced address ranges in frequently accessed regions (thus an increased number of ranges and tracing iterations is required). Nevertheless, these measures do not prevent consecutive data transfers from overflowing the ITM buffer. If an overflow does happen, ITM notifies about it, and the invasive approaches discussed below can be utilized to generate a trace without gaps due to overflows.

When exploring various invasive approaches, we observed that in some SoCs (e.g., TI CC2650) it is not feasible to clock CPU at a frequency low enough that the bandwidth of SWO is never exceeded. Therefore, we decided to simulate the behavior of ETM, which stalls the CPU when the output buffer overflows. We do it by configuring DWT to trigger a *DebugMonitor* exception when it emits a trace packet. This exception is handled by a FrankenTrace routine that busy-waits for a while to delay the next data transfer. For instance, tracing on TI CC2650 requires a delay of ~1000 cycles not to exceed the SWO bandwidth, as some transfers might generate up to two DWT tracing packets at once.

This approach makes the trace invasive, as the delays might change the flow of execution in timing-dependent software. For instance, a loop busy-waiting until a component's register changes its value could be executed fewer times in this scheme, as the component can make progress while the debug exception is being handled. Additionally, the recorded timestamps are no longer straightforwardly comparable across tracing iterations. However, the correct timing of such a loop can be learned from the noninvasive PC trace.

Our usage of the *DebugMonitor* exception has a few limitations. First, ARMv7-M defines multiple levels of execution priority, and specifies that an execution with a given priority can be preempted only when an exception with a higher priority arrives. By default, all configurable-priority exceptions and external interrupts share the same priority level. This is a problem, because if a matching data transfer is issued by the handler of an exception with the same priority as *DebugMonitor*, the FrankenTrace debug exception does not preempt the handler and thus does not delay the next transfer. However, the problem can be solved by modifying the traced software to lower the priorities of all exceptions below the priority of the debug exception.

Another problem is critical sections (*atomic* code): the debug exception does not interrupt their execution. In ARMv7-M, critical sections are implemented by temporarily raising the current execution priority to the highest one (commonly referred to as disabling exceptions) by changing the value of a CPU's special register (`PRIMASK`). However, ARMv7-M defines a method to effectively raise the execution priority to any level by writing to the `BASEPRI` special register. By modifying critical sections of the traced software so that they raise their priority by changing `BASEPRI` instead of `PRIMASK`, one can lower priorities of critical sections below the priority of the debug exception.[3] This way, critical sections remain mutually exclusive, but the debug exception can interrupt them.

However, one needs to be careful when modifying the execution priority of critical sections. The FrankenTrace exception routine adds only a delay, so in most circumstances it is safe to execute. However, in some cases the routine might cause a fault because of hardware restrictions (e.g., TI CC2650 cannot execute code from the main flash memory while switching the frequency source to another oscillator). For this reason, we do not modify the critical sections of low-level hardware drivers.

5.3 Reconstructing a Trace

To trace a single range of addresses, two DWT comparators are required: one emits a tracing packet, the other triggers the debug exception. TI CC2650 features four comparators, so only two address ranges may be traced during a single iteration, and because of the invasively introduced delays, the generated traces cannot be straightforwardly merged with each other by comparing transfers' timestamps. However, a single trace conveys an order between transfers within two address ranges. The order of all transfers can be reconstructed by tracing multiple pairs of ranges and combining the obtained partial orders. For instance, when tracing transfers within three address ranges (`A`, `B`, `C`), to infer their total order one needs to trace three pairs (for address ranges: `A+B`, `A+C`, `B+C`).

The resulting LSU trace can be combined with a PC trace by matching PC values of transfers in the former with PC values at consecutive cycles in the latter. Moreover, DWT can be configured to emit tracing packets when an execution of the FrankenTrace's exception routine starts and ends. This information can be exploited to assess timestamp differences between the LSU trace and the PC trace, and thus facilitate combining them into a full trace. Optionally, noninvasive LSU tracing may be performed (i.e., without triggering the debug exception). It would result in gaps in the trace each time the ITM buffer overflows, but it would yield correct timestamps and thus could further help align the LSU trace with the PC trace.

[3]Critical sections implemented with `mov rn, 1; msr PRIMASK, rn`, can be easily patched by just by replacing `PRIMASK` with `BASEPRI` to execute the sections with priority `1`, since the debug exception is executed with priority `0`.

6 INTERFACING WITH THE TPIU

We applied FrankenTrace to trace software run on TI CC2650 of a 1KT's CherryMote device. In doing so, we evaluated several hardware options of capturing tracing packets output on SWO.

Tracing a single SoC: In our main experiments, we employed a USB logic analyzer based on a popular Cypress CY7C68013A chip [9] (the same chip as used in HATBED), which allows for reliable sampling at up to 24 MHz. For robust symbol recovery we sample 3 times per symbol, achieving a maximum effective UART symbol rate of 8 Mbaud. The captured data is then processed in software by two decoders from the open-source sigrok project:[4] first by a UART decoder and then by an ARM ITM decoder. We modified the latter to save the decoded tracing packets to a CSV that we subsequently process to generate the final trace.

Scaling to a testbed: To assess whether FrankenTrace could be affordably deployed on a large scale, we first tested whether CherryMotes' low-cost supervising nodes, which accompany each TI CC2650, could be directly used to capture the tracing packets. Unfortunately, their built-in UART is not capable of reliable communication at the required speeds. However, CherryMote features an external USB port, which can host an FTDI FT232RL [8] UART-USB dongle. Preliminary experiments indicate that this allows for reliably receiving data over UART at 2 Mbaud, and that the effective tracing speed is limited only by the performance of the supervising node. This leads us to believe that FrankenTrace could be applied to many existing devices which feature low-speed debug components for as low as $10 (a cost of the UART-USB dongle), or incorporated into new designs for $5 (a cost of the FT232RL chip). Moreover, the M3 nodes of IoT-LAB already feature even faster UART modules (advertised at 12 Mbaud) and more performant supervising nodes, which leads us to believe that FrankenTrace can be easily deployed on IoT-LAB without any hardware modification.

7 CASE STUDY

To demonstrate the potential of FrankenTrace's PC and LSU tracing, we ran on 1KT's CherryMote an application that communicates over a low-power radio and blinks an LED when a packet is received, and traced its execution to analyze how it interacts with the SoC's peripherals. The tracing took 35 s (incl. the sleep intervals of the application) and recorded ∼140,000 CPU cycles and ∼2,500 data transfers. For comparison, invasive tracing done by stepping through subsequent instructions with a JTAG debugger running on 1KT's supervising node would require approximately 10 hours.

An LSU trace can be used to analyze accesses to memory-mapped registers of an SoC's components, for instance, to determine the values stored in those registers that affect the device configuration. An LSU trace can be simply searched for the addresses of the registers of interest. In contrast, deriving them from a source code analysis may not be easy, as the addresses for some memory accesses and the written values may not be known statically, and a given register can be accessed multiple times. In the sample program, we looked for register `DOE31_0`, whose bits correspond to enabling output on GPIO pins. We saw several accesses to the register in the LSU trace. The value of the last write was `0x00106000`, which meant that output was enabled on pins 13, 14, and 20.

As an interesting application of PC tracing, in turn, we selected timing analysis of busy-waiting loops. Such loops are used, for example, to wait for a memory-mapped register to change its value. In the considered program, we were able to identify such a loop using the LSU trace: in the trace, there was a write to the `CMDR` register, followed by two consecutive reads of the `RFACKIFG` register for the same PC value. The `CMDR` register is used to issue commands to the low-power radio (RF core) present in TI CC2650, and the `RFACKIFG` register indicates whether the RF core has acknowledged a command. Our finding indicated that the `RFACKIFG` register was read in a busy-waiting loop. Looking up the PC value of the `RFACKIFG` read in the PC trace and the program binary, we learned that indeed there was a busy-waiting loop that waited for `RFACKIFG` to be non-zero, and that the loop executed for around 3,168 cycles (66 μs).

8 CONCLUSIONS

We believe that FrankenTrace unlocks the potential of widely available low-speed ARMv7-M debug components. Its source code, along with a complete usage example, is available in our repository.[5]

For future work we envision testing FrankenTrace in more hardware setups (such as IoT-LAB's nodes), and revisiting the trade-off between invasiveness, trace completeness, and tracing time. For example, an online adaptive selection of tracing parameters (such as size of frequently accessed address ranges and the delay duration) could reduce overall tracing time and invasiveness of LSU tracing. Moreover, FrankenTrace could be extended to obtain more information-rich traces by exploiting other features of the low-speed debug components. For instance, DWT counters and comparators can be employed to trace the duration of multicycle instructions and instruction fetch stalls, providing an insight into the internal operation of Cortex-M's pipeline.

ACKNOWLEDGMENTS

The work presented in this article was supported by the National Science Center (NCN) in Poland under grant no. 2019/33/B/ST6/00448.

REFERENCES

[1] 2023. Orbcode. https://orbcode.org/.
[2] Cédric Adjih, Emmanuel Baccelli, Eric Fleury, Gaetan Harter, Nathalie Mitton, Thomas Noel, Roger Pissard-Gibollet, Frédéric Saint-Marcel, Guillaume Schreiner, Julien Vandaele, and Thomas Watteyne. 2015. FIT IoT-LAB: A Large Scale Open Experimental IoT Testbed. In *Proc. IEEE WF-IoT '15*. IEEE.
[3] Paramasiven Appavoo, Ebram Kamal William, Mun Choon Chan, and Mobashir Mohammad. 2018. Indriya2: A Heterogeneous Wireless Sensor Network (WSN) Testbed. In *Proc. TridentCom '18*. Springer, Cham.
[4] ARM. 2018. ARMv7-M Architecture Reference Manual, Issue E.d.
[5] ARM. 2023. External Trace Width and Bandwidth. https://perma.cc/6FQS-CD4U.
[6] Mateusz Banaszek, Wojciech Dubiel, Jacek Łysiak, Maciej Dębski, Maciej Kisiel, Dawid Łazarczyk, Ewa Głogowska, Przemysław Gumienny, Cezary Siłuszyk, Piotr Ciołkosz, Agnieszka Paszkowska, Inga Rüb, Maciej Matraszek, Szymon Acedański, Przemysław Horban, and Konrad Iwanicki. 2021. 1KT: A Low-Cost 1000-Node Low-Power Wireless IoT Testbed. In *Proc. MSWiM '21*. ACM.
[7] FTDI. 2019. FT232H Single Channel Hi-Speed USB to Multipurpose UART/FIFO IC Datasheet.
[8] FTDI. 2020. FT232R USB UART IC Datasheet.
[9] Infineon (Cypress). 2021. EZ-USB™ FX2LP USB 2.0 Peripheral Controller.
[10] Texas Instruments. 2016. CC2650 SimpleLink™ Multistandard Wireless MCU Datasheet.
[11] Li Yi, Junyan Ma, and Te Zhang. 2019. HATBED: A Distributed Hardware Assisted Testbed for Non-Invasive Profiling of IoT Devices. In *Proc. CPS-IoTBench '19*. ACM.

[4] https://sigrok.org – an open-source signal analysis software suite

[5] https://github.com/mimuw-distributed-systems-group/frankentrace