Boosting SW Development Efficiency with Function Lifetime Diagrams

Christoph Hazott Daniel Große
Institute for Complex Systems, Johannes Kepler University Linz, Austria
christoph.hazott@jku.at daniel.grosse@jku.at

***Abstract*—Embedded systems play a crucial role in today's *Internet-of-Things* (IoT) ecosystems. These systems can range from simple sensors to edge *Artificial Intelligence* (AI) solutions. However, their complex *Hardware* (HW)/*Software* (SW) interactions demand new analytical methodologies which encompass both the HW and the SW execution.**

In this work, we present a novel approach for early visualization of complex HW/SW interactions during SW development for embedded systems. Our approach traces the lifetime of HW and SW functions during the simulation of a *Virtual Prototype* (VP), which represents the HW while executing the SW. We dynamically instrument the execution of the VP at runtime such that neither the VP binary file nor the SW binary file has to be modified for tracing. The results are presented as a *Function Lifetime Diagram* (FLD) by storing the data into the *Fast Transaction Recording* (FTR) file format, which can be visualized, e.g. by the Surfer waveform viewer.

To demonstrate the effectiveness of our approach, we first analyze the HW and SW interactions of a *Micro-Electro-Mechanical System* (MEMS) sensor. More specifically, the root causes of two already identified HW/SW interaction issues are analyzed. Second, the application flow of an edge AI application for recognizing handwritten digits on a touch display utilizing a pretrained *Neural Network* (NN) is analyzed. These experiments demonstrate that FLDs provide an effective abstraction to foster a deeper understanding of the embedded system behavior. An additional runtime evaluation reveals an approximately 1.9-fold runtime overhead, demonstrating that our instrumentation approach remains runtime-efficient even for larger IoT applications.

I. Introduction

In the realm of embedded systems for *Internet-of-Things* (IoT), balancing real-time requirements, minimizing power consumption, and ensuring robust performance in diverse applications requires complex interactions between *Hardware* (HW) and *Software* (SW). To address these complex interactions at early design stages, *Virtual Prototype*s (VPs) are widely utilized when developing modern embedded systems SW [1]. This approach facilitates simultaneous SW development and HW architectural exploration.

In essence, a VP is a high-level executable model of the entire HW platform (processor, accelerator, sensor, bus, display, ...) which runs unmodified production SW [1]. As such, a VP allows simulation and validation of the functionality of an embedded system before physical implementation. For VP development, the most commonly used language is SystemC, a standardized C++ library (IEEE 1666 [2]); for an overview of SystemC, we refer to [3], [4]. SystemC includes an event-driven simulation kernel that serves as the foundation of the VP and is compiled into a simulation binary. Additionally, leveraging *Transaction Level Modeling* (TLM) [5], [6], e.g. for bus communication, enables a simulation performance that is orders of magnitude faster than *Register Transfer Level* (RTL) simulations. This significant speed-up is achieved by TLM through abstracting away unnecessary communication details while preserving the HW behavior. A prerequisite is to ensure the correctness of the VP, which has been addressed, for instance, in [7], [8]. Furthermore, using the VP for validation of SW has been considered, for example, in [9]–[13]. Altogether, a VP allows the SW developer to write and test code as if they were working directly on the actual HW platform, with full visibility of bus transactions and access to all HW registers.

However, in the process of SW development for an embedded system, the SW must leverage the HW, resulting in the aforementioned complex HW/SW interactions. These interactions are shaped by factors such as concurrency, interrupt handling, memory-mapped I/O, etc. As an example consider a *Micro-Electro-Mechanical System* (MEMS) sensor: the SW developer has to create sophisticated algorithms for sensor data processing, calibration and fusion while interacting with the underlying HW. Another example demonstrating the complexity of HW/SW interactions is the embedding of edge *Artificial Intelligence* (AI) applications utilizing frameworks like *TensorFlow Lite* (TFLite)[1] from Google [14]. Here, the SW developer must ensure that inputs, such as those from a touch sensor, and outputs, like those on a display, are correctly managed while the trained AI model is running. In general, keeping track of and visualizing HW/SW interactions necessitates opting for the right level of abstraction to focus on the relevant aspects while filtering out unnecessary complexity.

The chosen level should flatten out the interplay between HW and SW while keeping a meaningful level of detail on both sides. To the best of our knowledge, no such approach has been proposed yet.

In this paper, we present a **novel approach for visualization of complex HW/SW interactions in VPs**. As a suitable abstraction for the visualization, we identified the invocation of *functions*. Please note, the term function used from here on encompasses both SW functions (executed on an *Instruction Set Simulator* (ISS) of a VP) and SystemC functions modeling the HW behavior. Following this idea, our approach cohesively traces the lifetime of HW and SW functions during SystemC simulation and eventually forms the so-called ***Function Lifetime Diagram* (FLD)**.

To realize the proposed approach, we leverage the observ-

[1]Google renamed TFLite to LiteRT in September 2024.

ability of VPs. In the following, we describe our approach and demonstrate its usefulness by analyzing the HW and SW interactions of two common IoT applications, including (1) a MEMS gyroscope sensor and (2) an edge AI application utilizing a pretrained *Neural Network* (NN) for handwritten digit recognition on a touch display. At the heart of our implementation, we perform dynamic runtime instrumentation on the VP binary. More precisely, our approach makes use of the open-source runtime manipulation system DynamoRIO [15]. By this, we fulfill a major requirement: neither the VP nor the (embedded) SW have to be modified. Hence, the provided visualizations reflect exactly the HW/SW interactions which are executed on the VP and later on the real HW, i.e. no extra code is inserted distorting function lifetimes. For the visualization, we leverage the extensible open-source waveform viewer Surfer [16] such that we can make the complete approach available as open source on GitHub[2].

In summary, the key contributions of this work include:

- novel approach for visualizing complex HW/SW interactions of VPs with a FLD
- non-intrusive cohesive tracing for the VP (representing the HW) and the SW
- implementation available as open-source on GitHub
- fully reproducible experiments, also made available on GitHub

II. Related Work

There have been several papers which analyze/trace the HW side of a VP (see [17]–[19]), i.e. logging for instance TLM functions like `b_transport` together with the transported payload. Furthermore, on the SW side there have been works targeting the execution flow of variables [20] or visualizing symbolic execution traces [21]. Clearly, the HW/SW interaction is missing. Commercial tools, e.g. Virtualizer from Synopsys, offer functionality to visualize the execution of SW functions, but lack the information about the corresponding HW functions. Additionally, the visualization and analysis cannot be modified as these tools are proprietary. A general challenge for all these approaches is a non-intrusive realization, i.e. neither the VP itself nor the SW may be modified. The paper [22] presented a non-intrusive approach, which only focuses on HW and additionally has performance limitations due to the dependency on GDB [23].

In contrast to all these works, our approach focuses on determining lifetimes of SW functions and HW functions cohesively. We extend the approach introduced in [24], [25], which performs dynamic runtime instrumentation for non-intrusive cohesive HW/SW tracing. The resulting implementation and experiments will also be made available as open source.

III. HW/SW Function Lifetime Visualization

In this section, we present the proposed approach for visualization of complex HW/SW interactions. In Section III-A, we start by discussing two typical cases of HW/SW interactions,

[2]https://github.com/ics-jku/function_lifetime_diagram

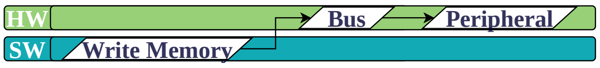


Fig. 1: SW executes *Write Memory*; HW identifies memory address as *Peripheral* target and transfers the data via *Bus*

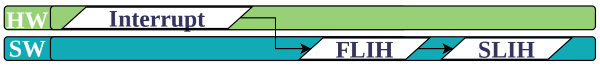


Fig. 2: HW *Interrupt* triggering the *FLIH* within the SW calling the *SLIH* to execute according actions

how they are taking place in a VP and why this ultimately leads to the consideration of function lifetimes to extract the HW/SW interactions. Thereafter, in Section III-B we describe our implementation.

A. HW/SW Interactions and Function Lifetimes

We want to specifically look at two cases of HW/SW interactions, which we use to illustrate our approach. First, we sketch both cases using two conceptual figures.

Fig. 1 depicts the first case where the SW writes into a register of a HW peripheral. In this case the HW/SW interaction depicted is based on memory-mapped I/O. First, the *Write Memory* of the SW executes e.g. the C/C++ line:

```
*CONFIG_REG_ADDRESS = 1;
```

Where `CONFIG_REG_ADDRESS` is a pointer containing the address of the configuration register of the HW peripheral. In this case, 1 is set to be the new value of the register. Next, the HW recognizes that the address belongs to the memory-mapped I/O region and transfers the data via the *Bus* to the target *Peripheral*, which then stores the new value in the configuration register.

The second case considers the triggering of an interrupt by the HW as depicted in Fig. 2. The interaction starts when the HW triggers an *Interrupt*. This trigger invokes the so-called *First-Level Interrupt Handler* (FLIH) within the SW. The FLIH has the task to quickly acknowledge the interrupt and prepare the system for subsequent processing, e.g. by storing the application context. After this, the FLIH invokes the corresponding *Second-Level Interrupt Handler* (SLIH), see right side of Fig. 2. The SLIH is registered to handle the specific interrupt to execute the appropriate SW actions, e.g. read peripheral data.

After the discussion of both figures, it becomes evident that they very well capture the HW/SW interaction in the sense that an action, e.g. triggering an interrupt in HW leads to a reaction, e.g. executing the according interrupt handlers in SW. In the context of VPs, action and reaction can be easily mapped to function executions: More precisely, due to the high-level nature of a VP, the behavior of the HW is modeled using **SystemC classes and methods; HW functions** in our terminology. Similarly, the **SW running** on the VP is modeled using **C/C++ classes, methods and functions; SW functions** in our terminology. For both, the **function information** can be easily extracted using the **debug information** generated by the compiler. Using this information, we get *where* a certain

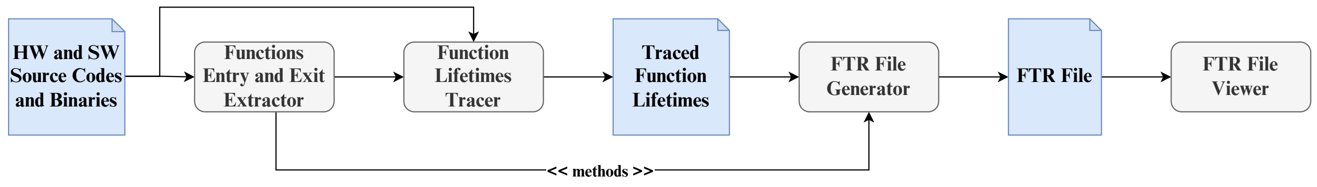

HW and SW Source Codes and Binaries
Functions Entry and Exit Extractor
Function Lifetimes Tracer
Traced Function Lifetimes
FTR File Generator
FTR File
FTR File Viewer
<< methods >>

Fig. 3: Function lifetime tracing, starting by extracting fuction information from HW/SW source code and binaries, tracing the desired entry and exit points and finally displaying the data; Blue boxes indicate input/output files for the flow stages which are depicted as gray boxes

functionality is executed, i.e. within the HW or the SW. Furthermore, the function name provides some information of *what* is executed. Missing is only *when* the functionality is executed to fully understand the HW/SW interactions. This information can be easily discovered by capturing the times when a function is entered and exited, or in other words the *function lifetime*. Note that for the SW (functions), running on the VP, the simulated program counter variable is utilized.

B. Function Lifetime Tracing

Our approach is based on dynamic runtime instrumentation as presented in [25]. The presented solution (implementation available on GitHub [26]) efficiently collects and evaluates structural coverage metrics of HW/SW executions in a cohesive manner.

In this work, we extended the existing solution by incorporating additional algorithms to enable tracing the lifetime of HW and SW functions, providing deeper insights into their interactions. The resulting tracing flow is depicted in Fig. 3. The flow starts with a so-called *Functions Entry and Exit Extractor*. This algorithm is parsing the source codes of the HW and SW to identify all existing function interfaces. If a function interface has been identified, the parser stores the position (file and line) of the function entry and determines the end of the function. Although a function has only one entry point, due to the possibility of exiting the function anywhere throughout its code block, multiple exit points can exist. Next, the extractor uses the debug information from the compiled VP and SW binaries to translate the identified code positions into *Program Counter* (PC) addresses. These addresses correspond to the memory locations where the stored instructions mark entry or exit points of the respective functions. To distinguish between the VP and the SW in subsequent discussions, we refer to the PC containing the VP position as **HW PC** and the PC containing the SW position as **SW PC**. The translated addresses are then given to the *Function Lifetime Tracer* which instruments the VP binary dynamically at runtime and is implemented as a DynamoRIO client. DynamoRIO is a runtime instrumentation system that exports an interface for building dynamic tools for a wide variety of applications. The concept behind DynamoRIO is to load and execute the binary, in our case the VP binary, within an application cache that can be manipulated at runtime. This mechanism allows for tracing of both, the HW and SW, while keeping the original binaries intact, eliminating the need for extensions to either. Moreover, provided that the simulation operates on a host PC compatible with DynamoRIO, contains debug information of the binaries,

```
1  static void trace_hw_pc(uint pc) {
2      (buffer->buf_ptr)->type = TYPE_HW;
3      (buffer->buf_ptr)->time_stamp = getTime();
4      (buffer->buf_ptr)->pc = pc;
5
6      buffer->buf_ptr += sizeof(buffer_entry);
7
8      if (buffer->buf_ptr >= buffer->buf_end) {
9          flush_file();
10     }
11 }
```
Listing 1: Function called by instrumentated code at runtime to record the occurance of a HW function entry or exit (with timestamp and pc) utilizing a buffer.

and simulates a type of SW PC, this method can be seamlessly applied.

The DynamoRIO client, created for our approach, utilizes the so-called *HW-to-SW memory hierarchy* to cohesively trace the HW and SW PCs. Listing 1 shows the implementation for tracing the HW PC. A call to this function is instrumented at the points where a HW function is entered or exited to write the information into the *Traced Function Lifetimes* file. To efficiently trace the SW, file accesses are reduced by storing the data in a buffer before they are written into the file (Line 2–4). This data contains the type `TYPE_HW`, which is a constant to indicate a HW function (Line 2), as well as the current timestamp (Line 3) and the content of the current HW PC (Line 4). As already mentioned, the HW PC contains the address of the HW function entry or exit point. Line 6 increments the buffer pointer to make space for the next occurrence of a HW PC. To make sure the buffer is not overflowing, Line 8–10 are flushing the buffer to the *Traced Function Lifetimes* at the time when the buffer is full. The implementation, tracing the SW PCs, looks quite similar. The difference in implementation arises because compared to the HW instrumentation, the SW PC is stored in a variable within the VP binary, therefore, only this variable needs to be traced leaving the SW binary unmodified even at runtime. The overall gathered data within the *Traced Function Lifetimes* file enables the generation of the FLD.

Next, an algorithm was implemented which reads the *Traced Function Lifetimes* file and combines the content with the original function information to generate the so-called *Fast Transaction Recording* (FTR) file. FTR was originally introduced to store TLM transactions [27]. These transactions are organized by streams which contain generators. Each of these generators contains a list of transactions with start and end times. By using streams for files and generators for functions, we can store our function executions in the form of transactions. The last step in the implemented flow is to visualize the gathered function lifetimes. Due to the use of the FTR file format, we are able to use different tools. For this work, we selected the Surfer waveform viewer [16].

IV. Experiments

As basis for our experiments, we use the open-source RISC-V VP++ [28]. We evaluate our approach for two applications running on this VP: First, we take the experiments from [25] which already identified two issues when adding a new MEMS gyroscope sensor peripheral to the VP. We analyze the function lifetime of these two issues to determine the root causes (Section IV-A). In the second part of our experiments, we consider the HW/SW interactions and application flow of a pretrained NN able to recognize a handwritten digit coming from a touch display connected to the VP (Section IV-B). In the last part of this section, the runtime impact of our approach is evaluated.

A. MEMS Gyroscope Sensor

With the experiments from [25], available as open source on GitHub [26], we analyzed the two published issues with our approach. The gyroscope sensor peripheral has a configuration register, a status register, and three data registers for x-, y-, z-axis, respectively. All these registers can be accessed using memory-mapped I/O via the generic TLM-2.0 bus. Additionally, the sensor peripheral is connected to the *Platform-Level Interrupt Controller* (PLIC) of the VP. To configure the sensor, the SW has four functions. The first function *init* is setting up the sensor via the configuration register. Further adjustments to the configuration should be made incrementally to preserve the previous settings. Three enable functions, one for each axis, called *enable_[x, y, z]*, are therefore designed to first read the configuration register, then set the bit for the corresponding axis, and finally write the updated configuration back to the register. The first issue involves incorrect register access behavior, while the second issue pertains to missing interrupts; in the following two paragraphs, we explain how the proposed approach helped to determine the root causes.

Register Issue: The register issue showcases a situation where the y and z register addresses are swapped. Such cases are hard to investigate because current approaches fail to cohesively capture HW reactions to SW actions, resulting in insufficient information being provided for in-depth analysis. In contrast, the visualization of the proposed FLD directly exposes this case. Fig. 4 shows an excerpt of the lifespan of the HW sensor peripheral via its SystemC `run` function and the SW via the `main` function. At the bottom of the FLD, we can see the execution of the `print_data` SW function to print the current sensor data for the aforementioned axes in the order x-, y- z-axis. The FLD shows that the HW registers of the sensor peripheral are accessed in the wrong order. The annotated bubbles ①, ②, ③ show that the order of access was x, z and y. This means the SW requested data from the z register instead of the y register and vice versa, leading to the root cause of misaligned register addresses in the SW.

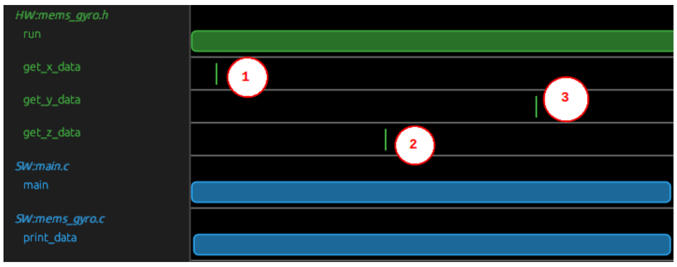


Fig. 4: Zoomed and annotated FLD for reading the axes' data registers, showing a wrong order for `get_y_data` and `get_z_data`; green is HW, blue is SW

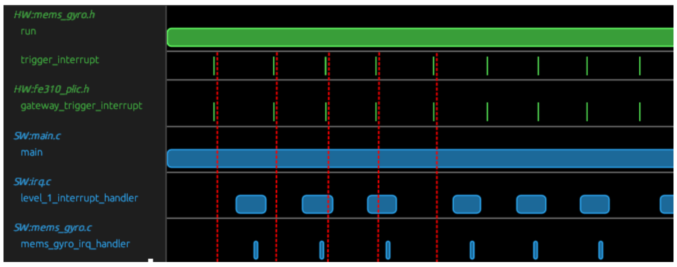


Fig. 5: Zoomed and annotated FLD for interrupt processing, showing a misallignment between the `trigger_interrupt` of the HW and the interrupt handlers of the SW; green is HW, blue is SW

Interrupt Issue: To signal the SW the availability of valid data, the sensor peripheral supports interrupts. The interrupt issue showcases a situation where the HW is triggering more interrupts than the SW can handle. Interrupts in general are hard to debug due to their asynchronous nature which can be altered when debug and tracing mechanisms are intrusive, as they can fundamentally influence the interrupt timings. So, having a non-intrusive approach which neither modifies the source code nor the binary is essential for interrupts. Fig. 5 depicts an excerpt of the FLD containing the interrupt handling. The image confirms that the execution order aligns with our expectations. The gyroscope HW (`trigger_interrupt`) tells the HW PLIC (`gateway_trigger_interrupt`) to execute the SW FLIH (`level_1_interrupt_handler`) which in turn executes the gyroscope SW SLIH (`mems_gyro_irq_handler`). The dotted red lines in the FLD mark the points at which an interrupt is triggered by the HW. Looking at the HW/SW interaction of the interrupt, we can see that the interval between being triggered by the HW and executed by the SW is growing for each interrupt over time (see bottom of Fig. 5). By the fourth interrupt, the FLIH is no longer executed (lost). This reveals the root cause: the SW cannot process interrupts at the rate they are triggered by the HW.

B. Edge AI Application

To evaluate the benefits of FLDs for understanding application flows, we consider a typical edge AI example: We want to recognize handwritten digits coming from a touch display. Therefore, we trained a NN with the widely recognized *Modified National Institute of Standards and Technology* (MNIST) database. As HW, we use the GD32 configuration of the VP which supports touch input and display output via a virtual display. For the SW part, we leverage TFLite from Google

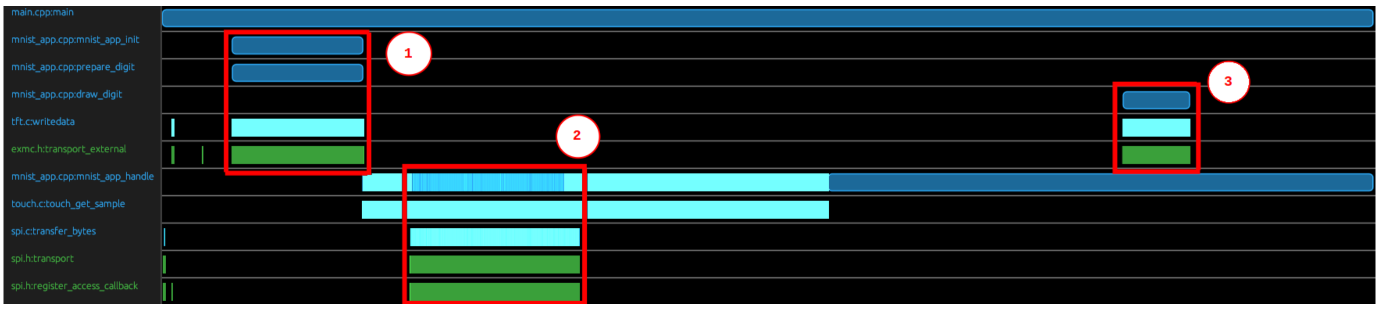
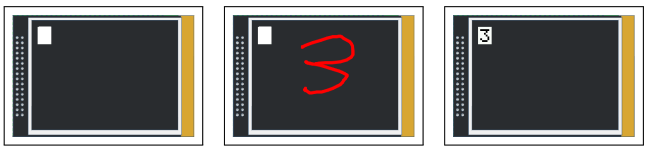

main.cpp:main
mnist_app.cpp:mnist_app_init
mnist_app.cpp:prepare_digit
mnist_app.cpp:draw_digit
tft.c:writedata
exmc.h:transport_external
mnist_app.cpp:mnist_app_handle
touch.c:touch_get_sample
spi.c:transfer_bytes
spi.h:transport
spi.h:register_access_callback
1
2
3

Fig. 6: Annotated FLD of single handwritten digit detection using NN pretrained with MNIST database; blue is SW, green is HW, boxes with border indicate single execution, boxes without border depict multiple executions, color tone indicates execution frequency (the brighter, the more frequent)



(a) Initialization (b) Touch Input (c) Display Output

Fig. 7: Digit recognition on GD32 from initialization over touch input to display output; white box indicates output area, black field indicates touch input area. Red digit is annotated to indicate touch input.

to deploy the trained NN. The user interaction is depicted in Fig. 7. In Fig. 7a, the application initializes the display. Next, the user draws a digit via touch on the display (Fig. 7b digit annotated in red). Finally, the input is processed with TFLite and the detected digit is printed on the display in the upper left corner (Fig. 7c). The pixel-by-pixel rendering of the display output, combined with the sampling of touch input, results in a high amount of HW/SW interactions. For these types of systems, it is crucial to have an abstraction layer that reduces unnecessary execution details while preserving the overall flow for effective analysis.

Fig. 6 shows the FLD generated when executing the application as depicted in Fig. 7. The collected function lifetimes are visualized in an order which best presents the previously mentioned application flow. Additionally, the diagram contains color coding which is needed to interpret the results. Boxes which have a border indicate that this is a single execution, e.g. the `main.cpp:main` SW function, which is executed only once, depicts the lifespan of the application. Boxes without a border illustrate multiple executions of the function, where the brightness of the color indicates the execution frequency (high brightness means high frequency). For example, the `mnist_app.cpp:mnist_app_handle` SW function begins with a bright blue box indicating frequent executions of this function. Within the annotated period ②, the box is slightly darker, indicating less frequent executions of this function. In this example, it is evident that the box is not filled with a uniform color but instead displays multiple executions of the function. With this in mind, we can interpret the FLD. The annotated numbers ①, ②, ③ show the segments in the diagram where the initialization ①, the touch input ② and the display output ③ take place.

Initialization: The diagram shows that for the initialization ①, the `mnist_app.cpp:mnist_app_init` SW function and the `mnist_app.cpp:prepare_digit` SW functions are executed. Additionally, the `tft.c:writedata` SW driver and the `exmc.h:transport_external` HW function for the external memory controller are invoked to draw the white field on the display. The high frequency of executions is attributed to the fact that the display output is rendered on a pixel-by-pixel basis.

Touch Input: The touch input is processed through the SPI interface and depicted by the area annotated with ②. We can see that on the SW side the `spi.c:transfer_bytes` function of the SPI driver is executed with a high frequency, polling the data from the HW. The polling is causing the HW functions `spi.h:transport` and `spi.h:register_access_callback` to collect the touch input via SPI and communicate the data back to the SW. For the regions before and after ② we see that the `mnist_app.cpp:mnist_app_handle` SW boxes are in bright blue until a certain point. The region before ② indicates the polling after initialization until the first touch input occurs. The region after ② is a wait time for some digits: Since the implementation is sensitive to the start of touch input, digits that require lifting the hand to write (e.g., 4) may otherwise be incorrectly recognized as two distinct digits. To solve this, a detection period was introduced linking multiple parts into one concise input. For touch input, the high frequency of executions is due to the sampling rate required to achieve the resolution necessary for the NN to accurately recognize the handwritten digit. We can additionally see a reduced execution frequency for the `mnist_app.cpp:mnist_app_handle` SW execution while the input occurs. This reduced frequency results from processing the input data.

Display Output: The execution of the `mnist_app.cpp:mnist_app_handle` SW function, after the detection period, contains the TFLite processing of the input data (box with border for `mnist_app.cpp:mnist_app_handle` SW function) and the output on the display ③. For this segment, we can see that the `mnist_app.cpp:draw_digit` SW function is invoked, which again invokes the `tft.c:writedata` SW function of the TFT driver and transports the information to the display via the `exmc.h:transport_external` HW function. Compared to the initialization, these HW/SW interactions are quite similar. The main difference is the initiator of the interaction, which were the `mnist_app.cpp:mnist_app_init` and the `mnist_app.cpp:prepare_digit` SW functions for the initialization and the `mnist_app.cpp:draw_digit` SW function for the display output segment. Again, the high frequency of execution is attributed to the pixel-by-pixel output.

C. Runtime Impact

Given the use of dynamic runtime instrumentation, in the final part of our experiments we evaluate the instrumentation impact of our approach by performing two types of runtime measurements.

The first type of measurement was conducted as a baseline measurement without instrumentation, while the second type was performed with instrumentation. Both measurements were carried out using the AI application as described in Fig. 7. Specifically, after initialization, the digit 3 was drawn manually as touch input, followed by the output of the recognized digit on the display. This process involves a significant amount of HW/SW interactions and consequently a high amount of instrumented code executions. Since this flow includes a manual input, which introduces variability, each measurement was repeated 10 times, and the average runtime was calculated to mitigate instability. We found that for our baseline measurement, the average runtime was about 4.38 seconds. The average runtime for the instrumented executions was about 8.20 seconds. By dividing the average of the instrumented measurement by that of the baseline measurement, we observe an approximate 1.9-fold increase in runtime when using our approach.

Overall, the experiments demonstrate that the FLD provides an effective abstraction for efficiently analyzing IoT applications while preserving detailed HW/SW interaction information, enhancing the understanding of embedded system behavior. Furthermore, the non-intrusive nature of the approach resulted in an acceptable runtime increase, making it also suitable for applications with a high amount of HW/SW interactions.

V. Conclusions

In this paper, we presented an approach for visualizing the intricate HW/SW interactions of embedded systems. Specifically, for VPs, which provide a high-level executable HW platform running unmodified SW, we have identified function invocations as suitable abstraction for extracting the HW/SW interactions. Our non-intrusive implementation (i.e. neither the VP binary nor the SW binary has to be modified) traces function lifetimes during VP simulation to generate a FLD.

In a first experiment, we considered SW for a MEMS gyroscope where already two issues have been reported. By visualizing the FLDs we were able to quickly determine the root causes. In a second experiment, we generated and analyzed the HW/SW interactions of touch inputs and display outputs as well as the application flow for an edge AI application recognizing handwritten digits on a touch display. Additionally, we have demonstrated that our approach introduces an acceptable runtime overhead by a factor of approximately 1.9.

For future work, we plan to extend our approach for multi-threaded SW applications as well as adding programmable analysis on top of the traces via concepts from [29]. Additionally, we will use the techniques from [30] to reduce the baseline runtime.

Acknowledgments

This work has been partially supported by the LIT Secure and Correct Systems Lab funded by the State of Upper Austria.

References

[1] T. De Schutter, *Better Software. Faster!: Best Practices in Virtual Prototyping*. Synopsys Press, March 2014.

[2] *IEEE Standard for Standard SystemC Language Reference Manual*, IEEE Std. 1666 (Revision of IEEE Std 1666-2011), 2023.

[3] V. Herdt, D. Große, and R. Drechsler, *Enhanced Virtual Prototyping: Featuring RISC-V Case Studies*. Springer, 2020.

[4] M. Hassan, D. Große, and R. Drechsler, *Enhanced Virtual Prototyping for Heterogeneous Systems*. Springer, 2022.

[5] *OSCI TLM-2.0 Language Reference Manual*, OSCI, 2009.

[6] F. Ghenassia, *Transaction level modeling with SystemC*. Springer, 2005.

[7] V. Herdt, H. M. Le, D. Große, and R. Drechsler, "Compiled symbolic simulation for SystemC," in *ICCAD*, 2016, pp. 52:1–52:8.

[8] ——, "Verifying SystemC using intermediate verification language and stateful symbolic simulation," *TCAD*, vol. 38, no. 7, pp. 1359–1372, 2019.

[9] V. Herdt, D. Große, H. M. Le, and R. Drechsler, "Early concolic testing of embedded binaries with virtual prototypes: A RISC-V case study," in *DAC*, 2019, pp. 188:1–188:6.

[10] M. Hassan, V. Herdt, H. M. Le, D. Große, and R. Drechsler, "Early SoC security validation by VP-based static information flow analysis," in *ICCAD*, 2017, pp. 400–407.

[11] V. Herdt, D. Große, J. Wloka, T. Güneysu, and R. Drechsler, "Verification of embedded binaries using coverage-guided fuzzing with SystemC-based virtual prototypes," in *GLSVLSI*, 2020, pp. 101–106.

[12] C. Hazott, F. Stögmüller, and D. Große, "Verifying embedded graphics libraries leveraging virtual prototypes and metamorphic testing," in *ASP-DAC*, 2024, pp. 275–281.

[13] ——, "Using virtual prototypes and metamorphic testing to verify the hardware/software-stack of embedded graphics libraries," *Integr.*, vol. 101, 2025.

[14] https://github.com/google-ai-edge/LiteRT, 2024.

[15] "Dynamorio," https://github.com/DynamoRIO/dynamorio, 2024.

[16] "Surfer," https://gitlab.com/surfer-project/surfer, 2024.

[17] M. Goli and R. Drechsler, "Through the looking glass: Automated design understanding of SystemC-based VPs at the ESL," *TCAD*, vol. 41, no. 4, pp. 1181–1185, 2022.

[18] W. Hong *et al.*, "Cult: A unified framework for tracing and logging C-based designs," in *S4D*, 2012, pp. 1–6.

[19] N. Bosbach, J. M. Joseph, R. Leupers, and L. Jünger, "NISTT: a non-intrusive SystemC-TLM 2.0 tracing tool," in *VLSI-SoC*, 2022, pp. 1–6.

[20] P. Pieper, V. Herdt, D. Große, and R. Drechsler, "Dynamic information flow tracking for embedded binaries using SystemC-based virtual prototypes," in *DAC*, 2020, pp. 1–6.

[21] J. Zielasko, S. Tempel, V. Herdt, and R. Drechsler, "3D visualization of symbolic execution traces," in *FDL*, 2022, pp. 1–8.

[22] M. Goli, J. Stoppe, and R. Drechsler, "Automated nonintrusive analysis of electronic system level designs," *TCAD*, vol. 39, no. 2, pp. 492–505, 2020.

[23] "GNU debugger," https://www.gnu.org/software/gdb, 2024.

[24] C. Hazott and D. Große, "DSA monitoring framework for HW/SW partitioning of application kernels leveraging VPs," in *IEEE DVCon Europe*, 2023, pp. 34–41.

[25] ——, "Relation coverage: A new paradigm for hardware/software testing," in *ETS*, 2024, pp. 1–4.

[26] "Relation coverage repository," https://github.com/ics-jku/relation_coverage, 2024.

[27] "Lightweight transaction recording for SystemC," https://github.com/Minres/LWTR4SC, 2024.

[28] M. Schlägl, C. Hazott, and D. Große, "RISC-V VP++: Next generation open-source virtual prototype," in *Workshop on Open-Source Design Automation*, 2024.

[29] L. Klemmer and D. Große, "WAVING goodbye to manual waveform analysis in HDL design with WAL," *TCAD*, vol. 43, no. 10, pp. 3198–3211, 2024.

[30] M. Schlägl and D. Große, "Fast interpreter-based instruction set simulation for virtual prototypes," in *DATE*, 2025.