Improving Software Testing Speed by 100X with SystemC Virtualization in IoT Devices

David Barahona, Motaz Thiab, Isael Díaz, Joakim Urdahl, and Milica Orlandic

Norwegian University of Science and Technology, davidbp.13@gmail.com
Norwegian University of Science and Technology, motaz.thiab@hotmail.com
Nordic Semiconductor, isael.diaz@nordicsemi.no
Nordic Semiconductor, joakim.urdahl@nordicsemi.no
Norwegian University of Science and Technology, milica.orlandic@ntnu.no

***Abstract*-The Internet of Things (IoT) industry has experienced exponential growth in recent years that has influenced more companies to incorporate non-traditional approaches during the design and development of their products to remain competitive. Virtual prototypes provide tools for improving productivity as well as reducing the time to market (TTM) and development cost of new products. However, this methodology has not been widely adopted in the realm of IoT for the development of embedded software. This article demonstrates that virtual prototypes can play a significant role in effective software testing for IoT solutions. In doing so, SystemC was used to implement a virtual prototype of a System on Chip (SoC) from Nordic Semiconductor, which is capable of executing several software tests. This virtual prototype provides a scalable and flexible solution that enables eliminating hardware dependencies during the embedded software development process. Results show that this methodology can speedup software execution close to 150 times, which represents a significant boost in the test design productivity. Furthermore, this methodology also provides enhanced observability of the hardware-software interaction.**

I. INTRODUCTION

A virtual prototype can be defined as a computer simulation of a physical product that can be presented, analyzed, and tested from concerned product life-cycle aspects such as design/engineering, manufacturing, service, and recycling as if on a real physical model [1]. From the Internet of Things (IoT) point of view, a virtual prototype can be seen as a high-speed software model of a physical hardware device. This system model often describes the system at a higher abstraction level while keeping the same functionality as the physical system. The use of virtual prototypes simplifies the hardware design space exploration and allows concurrent hardware and software development. All of which can help to reduce the time to market (TTM), save resources, find and fix bugs earlier, improve the test coverage, and even the teamwork between hardware engineers, software developers, and testers [2]. Nowadays, SystemC is the IEEE standard to develop these kinds of models, and it is also the most popular approach.

Even though virtual prototyping is not something new, this methodology has not been widely adopted in the IoT industry. This paper demonstrates that virtual prototypes can be a great resource during the development cycle of IoT products by dramatically speeding up the embedded software testing process. The benefits of virtual prototypes in IoT software testing are explored through the implementation of a virtual device that models a System on Chip (SoC) from the company Nordic Semiconductor.

This paper will first describe the software testing methodology at Nordic Semiconductor so the reader can identify the role the virtual prototype can play in this testing scheme. Then, an overview of the main components of the virtual prototype will be provided, details of their role in the system, and their level of abstraction will also be discussed. The results section will provide quantitative results for the execution time speedup of the tests that were selected to run on top of the virtual prototype and will also cover how the virtual prototype can improve the hardware observability during the software execution. Limitations of this approach are also discussed, as well as other challenges modern designs face when using virtual prototypes during the product design flow.

II. Software Testing Methodology

Embedded software in IoT applications is usually complex. For example, the software stack that makes it possible to implement several wireless communication protocols in Nordic Semiconductor's SoCs require thousands of lines of code to interact with peripherals like clock controllers, timers, counters, radio, among others. Due to this complexity, each component of the software stack requires an extensive testing process before being deployed as a part of real applications. Fig. 1 shows an overview of the process Nordic Semiconductor follows to test this kind of software. The approach makes a distinction between two different tests, unit tests and target tests, based on their goal and on the platform in which they execute.

A. Unit Tests

Unit tests are simple and have a scope usually limited to a single source file. These tests execute on a host PC, which means they do not require a physical target device to run. Unit tests exist mostly to evaluate the logic of the software. When these source files need to perform activities that demand some action from the hardware, empty mock functions come into play so the code can continue its execution. These mock functions have the same name but different implementations from the ones that interact with the hardware when the source code runs in an SoC.

B. Target Tests

Target tests have a broader scope than unit tests, and they exercise specific behaviors that involve one or more source files. Target tests, unlike unit tests, do not run on a host PC and require a physical connection to a target device. They execute on top of development boards that hold extra circuitry for peripheral access and debugging. Since target tests run on real boards, they need to compile and flash the program into the device memory to run as a regular application would. Two development boards are required to execute target tests. One of the boards becomes the device under test (DUT), and the other becomes a tester. They communicate through a radio channel and coordinate the test control using the UART peripheral.

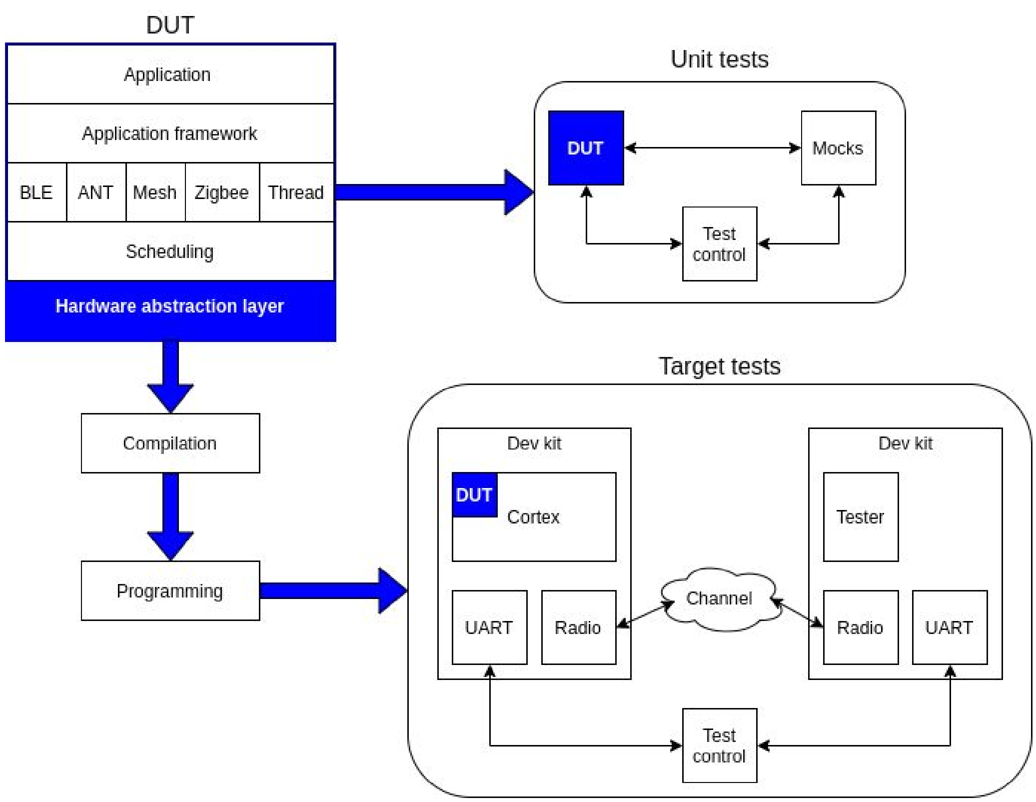

Figure 1. Embedded software testing methodology at Nordic Semiconductor

C. Where Does the Virtual Prototype Fit?

Using the methodology shown in Fig. 2 as a reference, the virtual prototype implementation focused on creating a new testing step between the unit tests and the target tests. The main goal is that the tests that require development boards can execute on the virtual device much earlier, even when the hardware is not yet available. This capability allows debugging most of the code in advance, so when the hardware is ready, only specific behaviors that require the boards need to be addressed. The target tests that were selected to run on top of the virtual prototype belong to the Multiprotocol Service Layer (MPSL) library from Nordic Semiconductor described in [3].

III. VIRTUAL PROTOTYPE ARCHITECTURE

In the context of IoT product development, a virtual prototype complexity and size depend on the features and functionalities that need to be tested. Given the tests that were selected to run on the virtual prototype, the system model design was not intended to cover all the peripherals available in the SoC but only specific peripherals. These include a real-time counter (RTC), timer, clock controller, temperature sensor, radio, and general-purpose I/O (GPIO). The virtual prototype also contains a CPU model and an interconnect element between the CPU and the peripherals, as shown in Fig. 2.

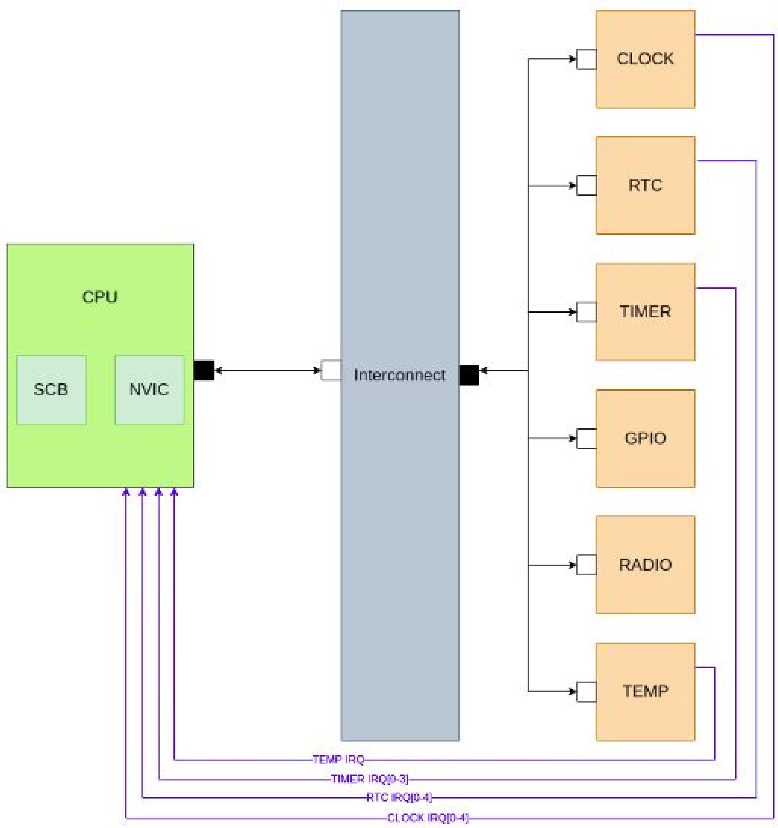

Figure 2. Virtual prototype overview

A. CPU

Even though the SoC uses an Arm-based CPU, this representation does not model its instruction set architecture, and instead, it provides a functional model of a CPU. The main goal of the model is to initiate transaction-level modeling (TLM) blocking read and write transactions to the different components of the system. Write transactions require the CPU to provide the address of the register as well as the write value. Read transactions demand only the register address. Fig. 3 illustrates a code snippet of the read and write transaction methods defined inside the CPU.

The CPU model also contains within its structure a System Control Block (SCB) and a Nested Vector Interrupt Controller (NVIC). Given that these components are part of the CPU and cannot be accessed by TLM transactions, the CPU possesses different methods to update the status of their registers. The CPU model is capable of detecting interrupt requests from the peripherals and perform interrupt service routines based on these requests.

```
void Cpu::apb_simple_write(int addr, int& data){
 apb_simple_transport(addr, data, 1);
}

void Cpu::apb_simple_read(int addr, int& data){
 apb_simple_transport(addr, data, 0);
}

void Cpu::apb_simple_transport(int addr, int& data, bool write){
 wait(clk.get_delay(2));
 tlm::tlm_generic_payload trans;
 sc_time delay = SC_ZERO_TIME;

 // Set simple transfer defaults
 trans.set_data_length(4);
 trans.set_streaming_width(4);
 trans.set_byte_enable_ptr(0);

 // Set transfer direction
 if(write){
  trans.set_command(tlm::TLM_WRITE_COMMAND);
 }
 else{
  trans.set_command(tlm::TLM_READ_COMMAND);
 }

 std::ostringstream ostr2;
 if(write){
        ostr2 << "Initiated write transaction of 0x" << std::hex << data << " to 0x"<<std::hex << addr;
        SC_REPORT_INFO("CPU", ostr2.str().c_str());
 }
 else{
        ostr2 << "Initiated read transaction to " << std::hex << addr;
        SC_REPORT_INFO("CPU", ostr2.str().c_str());
 }

 // Set addr and data ptr
 trans.set_address((sc_dt::uint64)addr);
 trans.set_data_ptr((unsigned char*)(&data));

 // Set target response variables to default
 trans.set_dmi_allowed(false);
 trans.set_response_status(tlm::TLM_INCOMPLETE_RESPONSE);

 // Start transport
 apb_init_socket->b_transport(trans, delay);

 // Check for response from target
 if(trans.is_response_error()){
  SC_REPORT_ERROR("CPU TLM-2:", trans.get_response_string().c_str());
 }
}
```

Figure 3. Read and write methods of the CPU model

B. Interconnect

The interconnect element acts as a bridge between the different components of the systems to create a channel for passing TLM transactions. In SystemC, an interconnect forwards and routes transaction objects between an initiator and a target. It is defined that only an initiator is capable of initiating TLM transactions. In this specific model, the CPU is the only initiator, and the peripherals act as targets. Transactions in the system are blocking transactions, which means that the CPU has to wait for a response from the peripheral before initiating a new one.

C. Peripherals

The modeling of the system peripherals was done based on Nordic Semiconductor's product specifications [4]. The abstraction level of each model depends on its use by the software tests that run in the virtual prototype. Therefore, only the functionalities required by these tests have been implemented. A brief description as well as implementation details of each of the peripheral models can be found below:

- Clock Control: the clock control peripheral is the one in charge of sourcing system clocks from a range of internal or external oscillators. It is also the one in charge of distributing them to the modules according to their specific needs. This peripheral has three fundamental purposes that are high-frequency control, low-frequency control, and low-frequency calibration. Given the nature of this component, the clock peripheral model is not based on SystemC clock classes but rather an ad-hoc implementation in which the model does not explicitly generate a clock signal. Inside the model, there are methods to check the availability of hardware resources, select the clock sources, and also others to trigger events and interrupts related to clock events.

- RTC: the RTC works as a simple timer that runs from the low-frequency clock source. In the peripheral model, when the counter increments its value, it is compared to the configurable capture registers. These registers can be configured using CPU transactions. When the capture register value matches the count value, the model triggers a compare event. Generating a compare event causes the model to set the value of an event register and also to trigger an interrupt in case these are enabled.

- Timer: the timer peripheral can act as a timer or as a counter based on the value of a mode register. When used as a timer, a prescaler is used to define the timer frequency based on two high-frequency clock sources. When used as a counter, the external source is given by a count event signal. For the purpose of the system model, it shares most functionalities with the RTC model, and they present only subtle differences. The timer peripheral does not have an accessible register with the counter value. Therefore, inside the timer model, it is just considered as an internal variable.

- Radio: the radio peripheral is composed of a 2.4 GHz radio receiver and transmitter. It also integrates modules to simplify direct memory access, automated packet assembly and disassembly, and also to automate cyclic redundancy check generation and checks. Given that tests that the virtual prototype is designed to execute almost do not require radio interaction, the radio peripheral is greatly simplified. The selected tests access only the power register to turn the peripheral on and off.

- Temperature sensor: the temperature sensor measures the die temperature and has a resolution of 0.25 degrees. Inside the peripheral model, after the CPU triggers a temperature measurement, an interrupt notifies the CPU that the process is complete, meaning that the result is ready to be read from the temperature register. The temperature measurement is modeled by waiting for some predefined amount of time. Also, since the die temperature is a physical quantity that cannot be obtained from running the test in the host CPU, it is modeled as a constant value.

- GPIO: the GPIO peripheral is made of digital pins that can be controlled by the user at run time. The GPIO model is designed to be able to set and clear individual bits at specific GPIO registers.

Fig. 4 provides the reader with an idea of how part of the behavior of a peripheral can be modeled. The figure illustrates a thread in charge of performing the temperature reading inside the temperature sensor peripheral model.

```
void nrf_temp::temp_thread(){
 while(1){
  wait(start_event); // Wait until measurement has started
  SC_REPORT_INFO("nrf_temp", "Temperature measurement started");
  wait(50, SC_US); // Simulate time to finish measurement
  if (isClr(regs.nrftempTASKS_STOP, 0x1)){ // Check if measurement was not stopped
   regs.nrftempTEMP = 42; // Return fixed temp value
   set(regs.nrftempEVENTS_DATARDY, 0x1); // Rise data ready flag
   datardy_event.notify(); // Notify the data ready event
  }
 }
}
```

Figure 4.Temperature read thread of the temperature sensor peripheral model

D. Models Verification

Each model in the system was verified individually before being deployed to the final design. The verification was based on simulations in which the CPU initiated a sequence of transactions to the model under test to recreate different scenarios to cover the complete model functionality. After verifying the system components individually, the whole system was also verified. Scenarios like triggering a temperature measurement based on a timer interrupt were exercised to ensure that the system components are capable of interacting between them.

IV. RESULTS

After implementing the virtual platform and being able to execute several target tests on top of it, execution times were measured to determine the speedup. Tests running in the development boards provide two execution times from the test output: the test execution time and the total execution time. The test execution time refers only to the time it takes for the test itself to execute, while the total execution time also includes the time it takes to erase the memory and flash the code. Since the virtual prototype only measures the time of the test execution, the test execution time from the development boards was the one used as a reference. Time measurements in the virtual prototype were obtained by inserting timing points in the top-level system constructor and destructor. Table I summarizes the results of the execution time comparison and the achieved speedup.

TABLE I
EXECUTION TIME COMPARISON

Test name	**Execution time (ms)**		**Speedup**
	Development boards	**Virtual prototype**	
MPSL init	1390	8.92	155.82
Temperature measurement	1384	9.55	144.92
One-shot Timer Callback	1535	10.28	149.31
One-shot Clock callback	1392	9.8	142
Clock callback started by protocol	1368	10.25	134

Results also show that the virtual prototype improves the observability of the interaction between hardware and software. The test output of the development boards is only able to provide information about the test name, status, and execution time. The virtual prototype, on the other hand, is capable of keeping track of every action that is being carried out by the hardware in a high level of abstraction. The virtual device can be configured to log the data and address of any register access, and also important events like turning a peripheral on or off, starting a counter, disabling an interrupt, among others. All the events are logged with a timestamp, meaning that a timing analysis can be performed if desired. Architecture design engineers can benefit from this observability in early design stages to find bugs or make architectural improvements. They can identify, for example, which register accesses are performed more often and modify the system to speed up these specific accesses. This kind of information can be used to make many important decisions before the SoC design is sent to the manufacturing facility, producing better performing devices.

V. LIMITATIONS

Even though results look promising for the virtual prototype implementation, this approach has some limitations that need to be considered. One limitation of virtual prototypes, in general, is that it is hard to prove complete equivalency between the real device and the prototype. Regarding this specific model, a limitation is that the CPU model is quite simple. The CPU model is not able to run the same code that runs on the development boards. The target tests require to replace all the hardware accesses to the boards with accesses to the virtual prototype. Including a full CPU model would allow compiling the same code that would run on the boards to run on the virtual prototype. This kind of model would improve the model accuracy, but it could also negatively impact the software execution speedup. Also, the software execution is not cycle-accurate due to the high abstraction level that the prototype represents. Once again, this implies a less accurate but faster simulation. Finally, it is also necessary to consider that there are certain behaviors that this prototype is not intended to model. Therefore, not all target tests can execute on top of the virtual prototype. For example, when using virtual prototypes to simulate a wireless link between two devices, there are physical aspects like noise and other conditions of the communication channel that were out of the scope of this work.

VI. CONCLUSION

This paper demonstrates that virtual prototypes are a powerful resource to speedup software testing in IoT. Eliminating the hardware dependency enables developing and testing code earlier, improving quality and reducing the TTM of new product designs. The virtual prototype also proved to deliver valuable hardware observability during software execution, which can allow teams in charge of the hardware architecture to use this data to improve their designs. Future work suggests the use of a commercially licensed CPU model, the development of new peripherals and the improvement of the existing ones, the consideration of other tests and libraries beyond the ones covered in this paper, and the inclusion of test automation features.

REFERENCES

[1] G. G. Wang, "Definition and Review of Virtual Prototyping ,"Journal of Computing andInformation Science in Engineering, vol. 2, no. 3, pp. 232–236, 01 2003

[2] T. De Schutter,Better Software. Faster!: Best Practices in Virtual Prototyping.SynopsysPress, 2014

[3] MPSL Library Internals (version 1.4.99)." [Online]. Available:https://developer.nordicsemi.com/nRF_Connect_SDK/doc/latest/nrfxlib/mpsl/doc/mpsl.html

[4] Nordic Semiconductor Infocenter(nRF52832)." [Online]. Available: https://infocenter.nordicsemi.com