Universiteit
Antwerpen

This item is the archived peer-reviewed author-version of:

Impact of software architecture on execution time : a power window TACLeBench case study

Reference:
Li Haoxuan, De Meulenaere Paul, Mercelis Siegfried, Hellinckx Peter.- Impact of softw are architecture on execution time : a pow er w indow TACLeBench case study
International journal of grid and utility computing - ISSN 1741-847X - 10:2(2019), p. 132-140
Full text (Publisher's DOI): https://doi.org/10.1504/IJGUC.2019.098216
To cite this reference: https://hdl.handle.net/10067/1579290151162165141

uantwerpen.be
Institutional repository IRUA

Copyright © 201x Inderscience Enterprises Ltd.

Impact of software architecture on execution time: a power window TACLeBench case study

Haoxuan Li

CoSys-Lab, IDLab, University of Antwerp,Antwerp, Belgium,
E-mail: Haoxuan.Li@uantwerpen.be

Paul De Meulenaere

CoSys-Lab, Flanders Make, University of Antwerp, Antwerp, Belgium,
E-mail: Paul.Demeulenaere@uantwerpen.be

Siegfried Mercelis

IDLab, imec, University of Antwerp, Antwerp, Belgium,
E-mail: Siegfried.Mercelis@uantwerpen.be

Peter Hellinckx

IDLab, imec, University of Antwerp, Antwerp, Belgium,
E-mail: Peter.Hellinckx@uantwerpen.be

Abstract: Timing analysis is used to extract the timing properties of a system. Various timing analysis techniques and tools have been developed over the past decades. However, changes in hardware platform and software architecture introduced new challenges in timing analysis techniques. In our research, we aim to develop a hybrid approach to provide safe and precise timing analysis results. In this approach, we will divide the original code into smaller code blocks, then construct a timing model based on the information acquired by measuring the execution time of every individual block. This process can introduce changes in the software architecture. In this paper we use a multi-component benchmark to investigate the impact of software architecture on the timing behaviour of a system.

Keywords: WCET; Timing Analysis; Hybrid Timing Analysis; Power Window; Embedded Systems; TACLEBench; COBRA Block Generator;

Reference

Biographical notes: Haoxuan Li obtained his M.Sc. in Biomedical Engineering from TUDelft. Later, he started Advanced Master Studies in Embedded Systems Design in ALaRI. In 2015 he started a PhD at CoSys-Lab and IDLab-imec at the University of Antwerp. His research focuses on timing analysis on real-time embedded systems.

Paul De Meulenaere is Professor of Automotive Engineering at the faculty of Applied Engineering of the University of Antwerp. His research is mainly oriented to software deployment onto embedded microcontroller platforms. In this area, software architectures such as AUTOSAR and OSEK are widely applied. He runs various research projects, often in collaboration with R&D divisions of mechatronic or automotive companies. He is also a member of Flanders Make, the Flemish research center for the mechatronics industry. Paul is spokesperson for the CoSys-Lab research group, which focuses on the design of embedded technology for cyber physical systems.

Siegfried Mercelis obtained his master degree in music production in 2008, followed by a master degree in applied engineering in 2012. From 2012 to 2016 he was employed at Van den Berghe R&D under a Baekeland PhD mandate on the subject of optimizing and parallelizing real time media applications. In December 2016 he obtained his PhD in Applied Engineering at the University of Antwerp, where he is currently employed as a postdoctoral assistant and member of the imec-IDLab research group in the team of Peter Hellinckx.

Peter Hellinckx obtained his Master in Computer Science and his Ph.D. in Science at the University of Antwerp. In 2009, he joined TERA-Labs at Karel de Grote University College where he became senior researcher and responsible for the distributed computing research group. In 2013, he became assistant professor in the faculty of applied engineering at UAntwerp. In 2015, he became head of the Electronics-ICT department and joined the IDLab research team as part of imec in 2016. He co-founded the spin-offs Hysopt and Hi10. His research focuses on distributed and real-time embedded software, cloud computing, IoT, cyber physical systems and agent based simulation.

1 Introduction

Real-time embedded systems are widely adopted in applications such as automotive, avionics and medical care. Many of these systems are required to meet strict deadlines. Therefore, obtaining the timing behaviour of such systems at the early design stage plays a crucial role in reducing the design cost and time-to-market. Timing analysis is the process of deriving or estimating the timing properties of a system. The ultimate goal of timing analysis is to predict safe and precise execution time bound[1]. Safe means the estimated time bound must cover the WCET (Worst-Case Execution Time). Precise means the estimated time bound is close to the actual WCET value.

1.1 Challenges in Timing Analysis

Different timing analysis techniques and tools have been developed over the past decades [2]. However, due to the increase of performance requirements and the intention of reducing the number of ECUs (Electronic Control Unit) used in one system [3], processors with more complex architecture components have taken over in industry [4]. The interactions between different tasks increases the complexity of timing analysis. The changes in hardware platform and software architecture introduced new challenges in timing analysis techniques. The traditional static analysis can no longer handle the analysis for complicated systems within a certain cost [5]. However, the lack of soundness creates hurdles when it comes to adopt measurement-based approach for safety critical systems [2].

1.2 Motivation

In our research, we are aiming to develop a hybrid approach that can overcome the challenges in both static and measurement-based approach. In this approach we propose to divide the original code into smaller code blocks then construct the complete timing model based on the timing information of every code block. However, subdividing the software into code blocks can be done in different ways. Additionally, the organization of these code blocks will change the software architecture of the system. Therefore, in order to understand the impact of software architecture on the execution time of a system, we conducted this research in which we measured the execution time of different software architectures composed with the same components from a multi-component benchmark [6].

1.3 Contributions

We created a multi-component powerwindow benchmark for the purpose of this study. The benchmark contains four main components including one driver-side window and three passenger-side windows. It is included in the TACLeBench [7] which is an available and comprehensive benchmark suite aiming at timing analysis. The detailed description about the benchmark is given in Section 2.

In this paper we investigate the impact of software architecture regarding to the end-to-end timing behaviour of the power window system. Section 3 provides an overview of the case studies. The results are elaborated and compared in Section 4 followed by discussion in Section 5. The conclusion is covered in Section 6

In Section 7 we propose a new hybrid timing analysis methodology which is aiming to offer soundness, flexibility and within a reasonable cost.

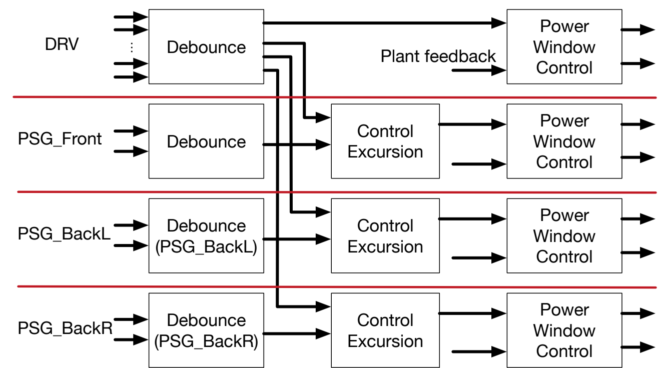

Figure 1 The complete scope of the power window Simulink model.

2 Powerwindow Benchmark

A power window is an automatic window which can be raised and lowered by pressing a button or a switch instead of a hand-turned crank handle. The complete application of a power window in a car usually consists of four windows which can be operated individually. The three passenger-side windows can be operated by the driver with higher priority. In addition to the basic functions, the power window model used in this paper also contains pinch force detection and end of the range detection[8].

2.1 Power Window Model

Figure 1 elaborates the complete power window Simulink model which consists of the driver-side window (DRV), front-passenger-side window (PSG_Front), back-left-passenger-side window (PSG_BackL) and back-right-passenger-side (PSG_BackR). The outputs of the driver-side debounce module are used as inputs by the control exclusion module of the passenger-side windows.

Since the three passenger-side power windows are identical, the front-passenger-side window is used for demonstration in this paper (Figure 2). Compared with the passenger-side power window, the driver-side power window excludes the control exclusion module and contains six more debounce circuits in the debounce module.

Copyright © 201X Inderscience Enterprises Ltd.

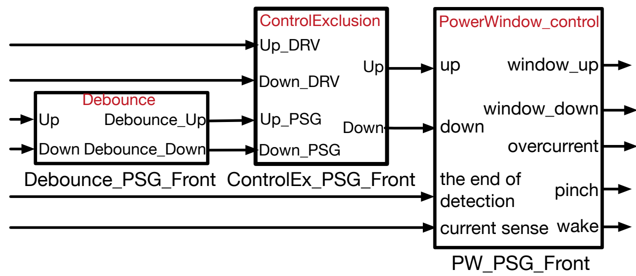

Figure 2 The front-passenger-side power window Simulink mode.

Debounce

When a push-button is pressed, it rebounds a bit before settling. This mechanical reaction often generates spurious inputs, for example, it may be read as multiple quick presses in a short period by the program. The debounce module (*Debounce_PSG_Front*) is implemented to filter out these faulty inputs. The debouncing function is triggered on the rising edge of a pulse signal with a period of 10 ms. Any signal that is shorter than 30 ms will not be forwarded to the next module. The debounce module is trivial because it guarantees the input only changes when the push-button is definitely pressed or released.

Control Exclusion

The control exclusion module (*ControlEx_PSG_Front*) is only included in the passenger-side windows. By using basic logic gates, the control exclusion module assigns higher priority to the driver when control inputs come from driver and passenger simultaneously.

Power Window Control

As the last module of the power window, the power window control module (*PW_PSG_Front*) takes the control signals from the control exclusion module (*up*, *down*) and the environment signals (*theendofdetection*, *currentsense*) to control the motor and the safety mechanism. Similar to the debounce module, the power window control function is connected to a pulse signal with a period of 50 ms and triggered on both the rising and falling edges. The input signals *the end of detection* and *current sense* are the feedback from the plant, which is a part of the environment.

2.2 Benchmark Description

The benchmark was initially generated by Matlab Embedded Coder from the Simulink model described in Subsection 2.1[6]. Modifications were made to convert the generated code to TACLeBench standards [7].

The input set was generated exclusively from the model in such way we increased the flexibility of the benchmark. Figure 3 illustrates the input set of the front-passenger-side window. The inputs were generated using Simulink Design Verifier with MC/DC (modified condition/decision coverage) coverage objectives[9], which is highly recommended for automotive.

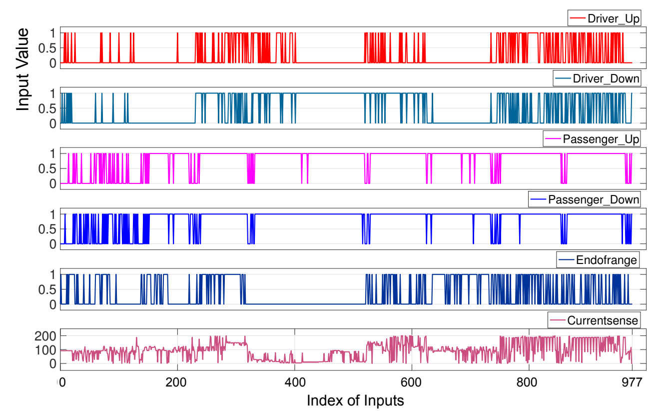

Figure 3 The front-passenger-side power window inputs.

The inputs were sampled every 5 ms, in totally 977 groups of inputs were included in the input set. During this research the input set was used for testing the front-passenger-side window as an individual benchmark. When testing on the complete power window benchmark, the *Driver_Up* and *Driver_Down* were connected to the driver-side power window. The cases covered by the input set include window position control from the passenger, simultaneous control signals from both the driver and the passenger, pinch detection, and end of the range detection.

The benchmark is selected mainly due to the flexible feature. The components of the benchmark can be easily converted into different configurations. It changes the software architecture but still preserves the functionality of the system, which is optimal to serve the purpose of our research.

3 Case Study

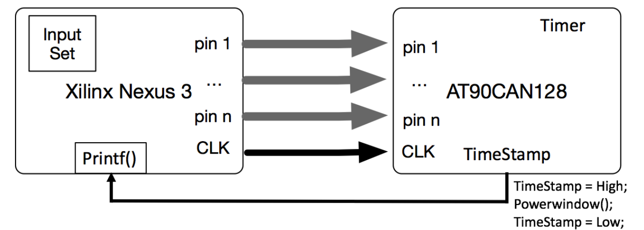

Figure 4 Experiment setup.

Figure 4 illustrates the experiment setup of this study. The powerwindow benchmark was deployed on the DVK90CAN1 development board which was embedded with an 8-bit Atmel AT90CAN128 single-core microprocessor. Due to the limited data memory of the processor, we introduced a Xilinx Nexus 3 FPGA to the setup. The inputs of the benchmark were sent from the FPGA to the processor. A time stamp pin was toggled before and after executing the benchmark

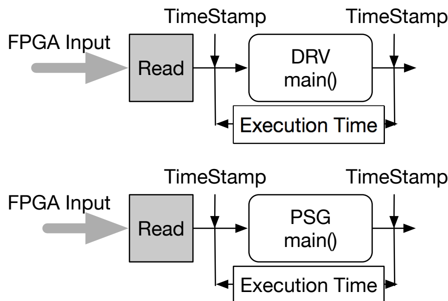

Figure 5 The implementation of case study 1.

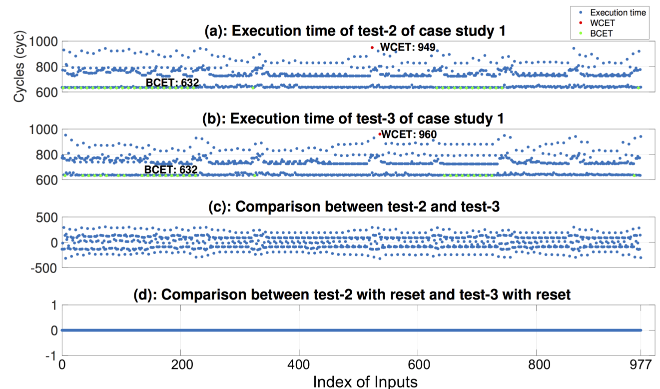

Figure 6 The results of execution time of case study 1 of PSG_Front.

function. The execution time was acquired by measuring the clock cycles between the time stamps. To assure the accuracy of the measured execution time, the processor was driven by the clock signal of the FPGA, in such way the processor was guaranteed to be synchronized with the FPGA. Eventually the results were sent through serial port to be collected for further analysis.

Technically speaking, every component of the benchmark can be considered as a periodic task. In this case, each task is triggered every 5 ms. In the beginning of each task, the processor reads the inputs from the input set. Once the measurement starts, the inputs from the FPGA also update every 5 ms. The update and read processes are synchronized to guarantee the inputs used by the benchmark is exactly the input we deployed on the FPGA.

In this study, one complete execution of the input set is considered as one test. The nth execution is represented by *test–n*. In total, the input set was executed 50 times consecutively in every case study.

During the case study we performed timing analysis on the benchmark in four cases:

1. driver-side and passenger-side windows execute in isolation as individual benchmarks.

2. four windows execute on one single-core processor in sequential order with interactions.

3. four windows execute as independent tasks with interactions under an operating system.

4. four windows execute as independent tasks with interactions under an operating system with different optimization levels.

Case study 1 was conducted to obtain the execution time of every power window when they are executed independently. The results of this case study were used as reference through the subsequent case studies. Case study 2 and 3 were designed to investigate the impact of the software architecture on the execution time when tasks are executed in sequential order and as independent tasks with interactions respectively. Last but not the least, case study 4 focused on the role of compiler optimization in timing analysis when the software architecture stays consistent.

4 Results

The execution time of the benchmark was acquired using measurement-based approach with the experiment setup described in the previous section. The results of the case studies are elaborated in this part of the paper.

4.1 Case Study 1

In case study 1 the driver-side window and passenger-side windows were tested as individual benchmarks (Figure 5). The benchmarks were executed consecutively without resetting the board or reinitialization of the power window function. The results of this case study are used as the reference through the whole study.

As showed in Table 1, the BCET (Best-Case Execution Time) of the driver-side window was reached at 1189 *cyc* (148.625 μs), the WCET was at 1986 *cyc* (248.25 μs). The results of the three passenger-side windows were identical, where the BCET and the WCET reached 632 *cyc* (79 μs) and 977 *cyc* (122.125 μs) respectively.

Figure 6:(a) shows the result of the first execution (*test*–1) of the complete input set. Figure 6:(b) is the result of the second execution (*test*–2) of the complete input set. The BCETs for both tests were at 632 *cyc*, whilst the WCETs were at 958 in *test*–1 and 949 in *test*–2. The BCET occurred more frequently than WCET. Although the case study was carried on a very simple single-core processor without any non-deterministic components such as cache and branch prediction, *test*–1 and *test*–2 still showed different timing behaviours. By subtracting the results of test-2 from test-1, differences with the maximum magnitude at approximate 250 *cyc* could be observed in Figure 6:(c).

During the experiment, the timing behaviour of the benchmark started to repeat after 10 consecutive tests (Figure 7). Identical execution time patterns occurred in test-2 and test-12, and test-22. The observation was consistent across all the tests except for test-1 (Figure 7:(a)).

An additional experiment was performed based on the results of this case study. The benchmark was

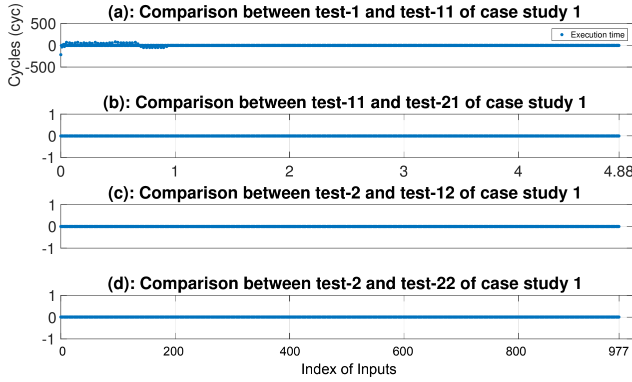

Figure 7 The comparison of execution time of case study 1 of PSG_Front cross tests.

reinitialized before every test to ensure it had the same initial state. The result can be seen in Figure 6:(d), in this case, the benchmark showed exactly same timing behaviours among the tests.

Table 1 The results of case 1.

Task	BCET (*cyc*)	BCET (μs)	WCET (*cyc*)	WCET (μs)
DRV	1189	148.625	1986	248.25
PSG_Front	632	79	977	122.125
PSG_BackL	632	79	977	122.125
PSG_BackR	632	79	977	122.125

DRV: driver-side window; PSG: passenger-side window; BackL: back-left; BackR: back-right; BCET: best-case execution time; WCET: worst-case execution time; *cyc*: system clock cycle

4.2 Case Study 2

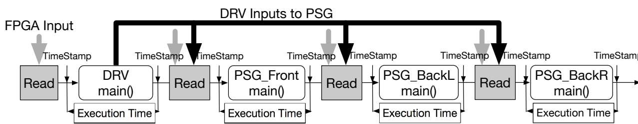

Figure 8 The implementation of case study 2.

In Case study 2 the four power windows were organised in sequential order (Figure 8). Thus the outputs from driver-side window were used by the passenger-side windows as inputs. The purpose of this case study is to investigate the impact on the execution time when the tasks are in sequential order.

The BCET and the WCET of the driver-side window showed the same results as in case study 1 (Table 2). The WCETs of the passenger-side windows were 991 *cyc* (123.875 μs), which were slightly higher compared with the 977 *cyc* observed in case study 1.

Figure 9 is the comparison between case study 1 and case study 2, where the driver-side window revealed identical timing behaviours in both case studies (Figure 9:(a)), whilst this observation did not hold for the

Table 2 The results of case 2.

Task	BCET (*cyc*)	BCET (μs)	WCET (*cyc*)	WCET (μs)
DRV	1189	148.625	1986	248.25
PSG_Front	632	79	991	123.875
PSG_BackL	632	79	991	123.875
PSG_BackR	632	79	991	123.875

DRV: driver-side window; PSG: passenger-side window; BackL: back-left; BackR: back-right; BCET: best-case execution time; WCET: worst-case execution time; *cyc*: system clock cycle

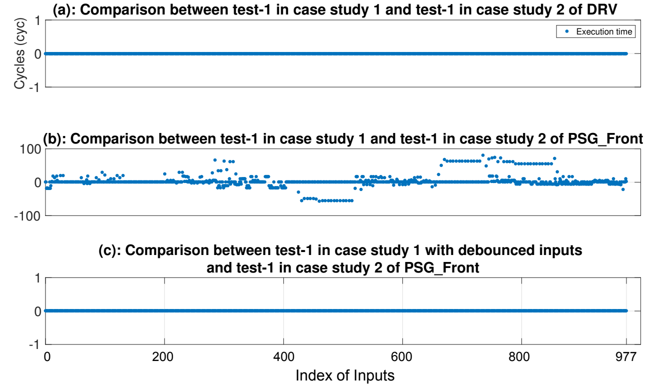

Figure 9 The results of case study 2.

passenger-side windows. Differences occurred between the *test*–1 in case study 2 and the *test*–1 in case study 1 of the passenger-side window (Figure 9:(b)).

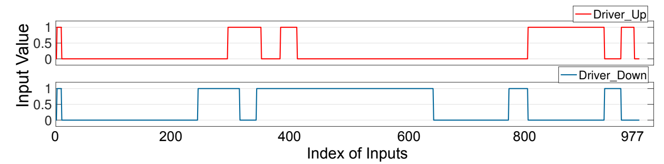

Figure 10 The debounced driver-side inputs for the front-passenger-side power window.

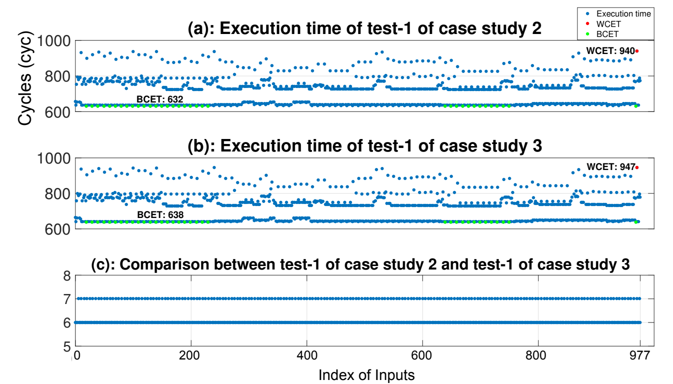

Figure 11 The results of PSG_Front in case study 3.

To inspect the reason for the difference in the measured WCETs of the passenger-side windows in case study 1 and case study 2, we modified the experiment

Table 3 The results of additional experiment in case study 2.

Task	BCET (*cyc*)	BCET (μs)	WCET (*cyc*)	WCET (μs)
PSG_Front	632	79	991	123.875
PSG_BackL	632	79	991	123.875
PSG_BackR	632	79	991	123.875

PSG: passenger-side window; BackL: back-left; BackR: back-right; BCET: best-case execution time; WCET: worst-case execution time; *cyc*: system clock cycle

with 2 major changes: first, an isolated passenger-side window benchmark was used as in case study 1; second, the input signals from the driver were debounced before entering the passenger-side power window. The debounced inputs are presented in Figure 10. The results showed identical behaviour compared with case study 2 (Table 3; Figure 9:(c)).

4.3 Case Study 3

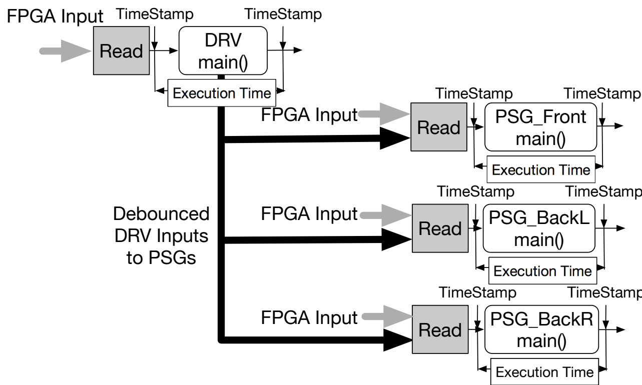

Figure 12 The implementation of case study 3.

Case study 3 investigated the impact on the execution time when tasks are executed independently with interactions under an operating system. The OSEK [10] operating system was employed to schedule the four windows (Figure 12). The tasks were defined as non-preemptable periodic tasks with a period of 5ms. All the tasks were assigned the same priority level. The arrival of the tasks was arranged such that the driver-side window could be executed prior to the passenger-side windows, which guaranteed the correctness of the inputs received by the passenger-side windows.

Figure 11:(a) is the execution time of the front passenger window in case study 2. The BCET and WCET were 632 and 940 *cyc* accordingly. Figure 11:(b) is the results of the same test in case study 3 where the BCET and the WCET increased to 638 and 947 *cyc*. By subtracting (a) from (b), the differences in execution time between the tests can be seen in Figure 11:(c), which is 6 or 7 *cyc*.

Additionally, we also tested non-preemptable tasks with different priorities and preemptable tasks with different priorities. The results were identical as demonstrated in Figure 11. It can be seen in Table 4,

Table 4 The results of case 3.

Task	BCET (*cyc*)	BCET (μs)	WCET (*cyc*)	WCET (μs)
DRV	1195	149.375	1992	249
PSG_Front	638	80.5	998	124.75
PSG_BackL	638	80.5	998	124.75
PSG_BackR	638	80.5	998	124.75

DRV: driver-side window; PSG: passenger-side window; BackL: back-left; BackR: back-right; BCET: best-case execution time; WCET: worst-case execution time; *cyc*: system clock cycle

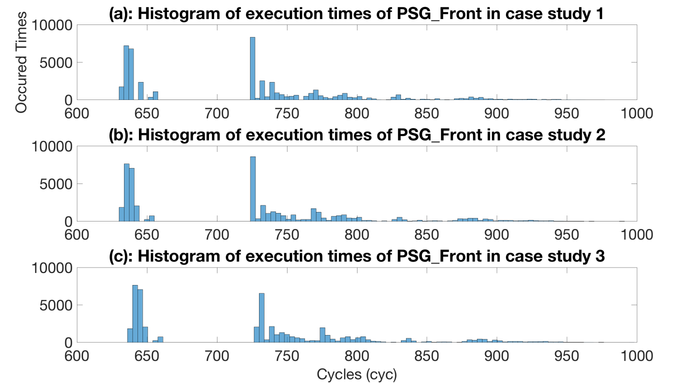

Figure 13 The histogram of execution times (from *test*–1 to *test*–50) of PSG_Front in 3 case studies.

the WCETs increased by 7 *cyc* and the BCETs increased by 6 *cyc*. The 3 passenger-side windows stayed identical with each other.

Figure 13 illustrates the execution times of the front-passenger-side window in histograms. It can be observed that in case study 1 and 2, most of the execution times were around 650 *cyc* and 750 *cyc*. Changes could be observed in case study 3, where the execution distribution moved slightly towards to the right. In all three cases, a few outliers resulted in much higher WCETs compared with the average execution time.

4.4 Case Study 4

In the above three case studies we used AVR Compiler 4.9.2 with optimization option -O1 as default settings. To further investigate the role of compiler in timing analysis, we designed case study 4.

In case study 4 we used AVR Compiler 4.9.2 at different optimization levels including *Optimize (-O1)*, *Optimize more (-O2)*, *Optimize most (-O3)*, *Optimize for size (-Os)*.

Figure 14 is the differences in execution time under different optimization options between case study 2 and case study 3. The figure was obtained by subtracting the results of case study 2 from case study 3. It can be noticed that with the increase of the optimization level, the gap reduced from 6 *cyc* or 7 *cyc* in -O1 to 2 *cyc* in -O3 .

Table 5 indicates the results of PSG_Front executing as independent tasks with interactions (as in case study 3). The shortest BCET occurred at -O2 at 593 *cyc*, the

Figure 14 The comparison of the execution time under different optimization options between *test*–1 in case study 2 and case study 3. -O1: Optimize; -O2: Optimize more; -O3: Optimize most; -Os: Optimize for size

Table 5 The results of case 4 (PSG_Front).

Task	BCET (*cyc*)	BCET (μs)	WCET (*cyc*)	WCET (μs)
-O1	638	79.75	998	124.75
-O2	593	74.125	899	112.375
-O3	609	76.125	868	108.5
-Os	626	78.25	916	114.5

-O1: Optimize; -O2: Optimize more; -O3: Optimize most; -Os: Optimize for size; BCET: best-case execution time; WCET: worst-case execution time; *cyc*: system clock cycle

longest WCET was 998 *cyc* at -O1. The biggest difference between the shortest BCET and longest BCET was 45 *cyc*. The biggest difference between the shortest WCET and longest WCET was 130 *cyc*.

5 Discussion

In case study 1 we observed that the execution time of the system showed variations between different tests and started to show repetitive behaviours every 10 consecutive tests. After an in-depth investigation, we noticed the loop observed in this case study was due to the initial state of local counter variables. The power window model we used to create the benchmark consists of state charts which are being triggered periodically. The local counter variables are created to keep the timing information for the state charts. Thus, it resulted in history-dependent behaviour of the powerwindow benchmark. Based on this study, we concluded the execution time of such systems is not only dependent on the inputs, but also on the precedent executions. In this case, it was the value left in the counter variables after the previous execution.

In the additional experiment the power window showed exactly the same behaviours after the benchmark was reinitialized before every test. This confirmed our previous assumption. Therefore, in order to estimate the execution time of a history-dependent system, it is necessary to identify the variables which have impact on the timing behaviour prior to the input generation.

In case study 2, after connecting the passenger-side windows with the driver-side window, different timing behaviours were observed in the passenger-side windows. The additional experiment proved the differences were due to the driver-side inputs first passing through the debounce function in the driver-side window before entering the passenger-side window. It indicates that the test cases generated by Simulink Design Verifier were not sufficient to reach the full coverage, especially the worst case scenario. Special cautions should be given when using Simulink to generate test cases for critical systems. Not only the external inputs, but also the variables which influence the execution time need to be included in the input set. For a simple processor as AT90CAN128, as long as the input is adequate, interactions between sequential tasks will not affect the WCET.

Additional investigation showed that the increase in the execution time in case study 3 was due to the combination of the introduce of OSEK operating system and the compiler. Because the OSEK operating system was compiled together with the benchmark, the compilation resulted in slight inconsistencies at assembly level. For instance, the subroutine call compiled to *RCALL* (relative call to subroutine) in case study 1 and case study 2 was compiled to *CALL* in case study 3. The former is a relative call to an address within PC (program counter) - 2K + 1 and PC + 2K (words), the latter calls to a subroutine within the entire program memory [11]. Timing-wise *RCALL* takes 3 *cyc* to execute whilst *CALL* requires 4 *cyc*. Hence, the change in the execution time was the aftermath of the differences in the assembly code.

The results in [6] showed more drastic increase in the execution time when using OSEK operating system compared with case study 3. There could be two reasons behind this scenario. First is the change of the compiler. In [6] all the other case studies used AVR Compiler 4.9.2, while in case study 3 AVR Compiler 4.3.3 was used. Second, by repeating case study 3 with the new experiment setup using AVR Compiler 4.3.3. We noticed all the outliers in [6] were 1731 *cyc* higher compared with the execution times at the same points obtained using the new experiment setup. Compared with the setup in [6], the setup of this study removed the serial communication between processor and computer when the power window reads the input signals. Therefore, although in the previous experiment we did not include the time for serial communication in the measurement, it still introduced overhead in the results.

In case study 3, the reason that the results were identical cross different settings of tasks was because the tasks were released at the same time, consequently, the low priority tasks were always scheduled to execute after the high priority task. Thus, preemption did not happen during the execution. To study the impact of the preemption, the input set must be modified so that even

after preemption, the inputs taken by the windows are still consistent compared with the other case studies.

Case study 4 shows the impact of optimization option on the execution time. The extra execution time was introduced for the same reason as explained in the previous text in all four optimization levels. The only difference was with the increase of the optimization level, the differences between the assembly code without OSEK operating system and the assembly code with OSEK operation system become more and more trifling. In combination with case study 3, we conclude that in order to assure the consistency between the behaviour of the complete code and the behaviour of the combination of the code blocks, it is necessary to analyse the timing behaviour at assembly level.

6 Conclusion

The results of this research suggested that for a simple real-time system running bare-metal on a processor without complicated architecture components, it is safe to analyse the timing behaviour of every component individually. When provided with sufficient input set, analysing components individually can reduce the complexity of the analysis process whilst still yields reliable results. Whereas an operating system is involved or the size of the complete system is substantial, some overhead may be introduced by the divergence at assembly level. However, since the increase of the WCET is rather small compared with the overall execution time, this overhead can be compensated by adding a safety margin to the results obtained from individual component. Significantly, to estimate the overall WCET of the complete system combined with operating system, it is possible to perform timing analysis separately on the functional system and the operating system. It will not only reduce the complexity of the analysis, but also save efforts spent in deriving the timing profile of the operating system repetitively.

7 Future Work

7.1 A Hybrid Approach for Timing Analysis

For the future research, we propose a new hybrid methodology for timing analysis (Figure 15). In step 1, we separate the complete C code into smaller blocks using COBRA Block Generator which is a tool developed to facilitate the process of generating a hybrid block model and corresponding source files for time measurements [12] by the IDLab in the University of Antwerp. Indication flags are set in front of C blocks before the compilation, so that the blocks can be traced and reconstructed on assembly level. Every obtained assembly block is correspondent to one C block at the C code level.

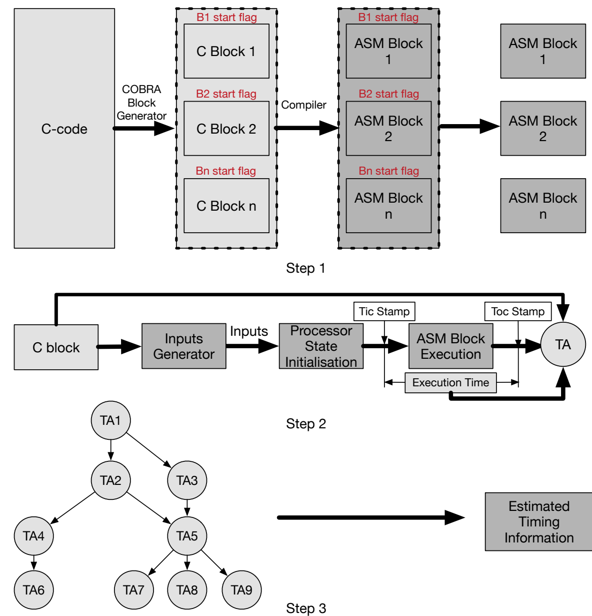

Figure 15 The proposed analysis methodology.

In step 2, the inputs generator creates inputs accordingly for every C block. The created inputs will be used in the processor state initialization process. During the processor state initialization, the stack and the registers of the processor will be adjusted so that the created inputs will be passed on correctly to the assembly block. After the processor is set to the desired state, the timing measurement is performed on the assembly code. The acquired execution times is then used to compose timed automata for the C blocks.

In step 3, after all the blocks obtained in step 1 are converted into corresponding timed automata, the timing model of the system can be constructed. Eventually, the timing information will be retrieved and analysed.

7.2 Motivation

Our previous results indicate that for measurement-based timing analysis, the inputs are of uttermost importance. However, a reliable and systematic input generation approach is still in high demand. Case study 1 and case study 2 showed that during input generation, crucial variable may be ignored during the process. Case study 3 and case study 4 suggested one major impact of software architecture on execution time is due to the divergence in assembly code. Therefore, we propose this methodology with two important features: first, dividing the code into smaller code blocks for input generation; second, performing timing analysis on the code blocks at assembly level. The former is aiming to provide a higher possibility to achieve the coverage of the worst-case scenario. The latter intends to eliminate the differences between the original code and the analysed code at assembly level.

7.3 Improvement

We expect that compared with the current measurement-based approach, the proposed methodology would be more reliable regarding to the worst-case scenario coverage. Compared with the traditional static approach, this approach should be able to reduce the cost of constructing a timing model for complicated systems especially under the boost of processors with advanced features. In comparison with the current state of the art hybrid approaches, the proposed approach should bring 3 major improvements: first, instead of executing the complete software using the inputs generated for the complete system to collect the timing information of one single block, this methodology can generate inputs respectively for every C block and perform measurement on every block independently; second, by measuring at assembly level, the consistency between the analysed code and the original code can be satisfied; third, the COBRA Block Generator facilitates block generation on different abstraction levels. As the size of the block is adjustable, it is possible to decide the granularity of the timing analysis. Thus gives the opportunity to choose between the soundness of the results and the cost of the analysis.

Acknowledgement

This research was partially supported by Flanders Make vzw, the strategic research centre for the manufacturing industry.

References

[1] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, et al. The worst-case execution-time problemoverview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3):36, 2008.

[2] Reinhard Wilhelm, Sebastian Altmeyer, Claire Burguière, Daniel Grund, Jörg Herter, Jan Reineke, Björn Wachter, and Stephan Wilhelm. Static timing analysis for hard real-time systems. In *Verification, Model Checking, and Abstract Interpretation*, pages 3–22. Springer, 2010.

[3] Nicolas Navet et al. Multi-source and multicore automotive ecus-os protection mechanisms and scheduling. In *Industrial Electronics (ISIE), 2010 IEEE International Symposium on*, pages 3734–3741. IEEE, 2010.

[4] Rohan Tabish et al. A real-time scratchpad-centric os for multi-core embedded systems. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–11. IEEE, 2016.

[5] Leonidas Kosmidis, Eduardo Quiñones, Jaume Abella, Tullio Vardanega, Ian Broster, and Francisco J Cazorla. Measurement-based probabilistic timing analysis and its impact on processor architecture. In *Digital System Design (DSD), 2014 17th Euromicro Conference on*, pages 401–410. IEEE, 2014.

[6] Haoxuan Li, Paul De Meulenaere, and Peter Hellinckx. Powerwindow: a multi-component taclebench benchmark for timing analysis. In *International Conference on P2P, Parallel, Grid, Cloud and Internet Computing*, pages 779–788. Springer, 2016.

[7] Heiko Falk et al. Taclebench: A benchmark collection to support worst-case execution time research. In *16th International Workshop on Worst-Case Execution Time Analysis*, 2016.

[8] Ken Vanherpen et al. Model transformations for round-trip engineering in control deployment co-design. In *Proceedings of the Symposium on Theory of Modeling & Simulation: DEVS Integrative M&S Symposium*, pages 55–62. Society for Computer Simulation International, 2015.

[9] Kelly J Hayhurst et al. A practical tutorial on modified condition/decision coverage. 2001.

[10] Andree Zahir et al. Osek/vdx-operating systems for automotive applications. In *OSEK/VDX Open Systems in Automotive Networks (Ref. No. 1998/523), IEE Seminar*, pages 4–1. IET, 1998.

[11] *AVR Instruction Set Manual.*

[12] Thomas Huybrechts, Yorick De Bock, Haoxuan Li, and Peter Hellinckx. Hybrid approach on cache aware real-time scheduling for multi-core systems. In *International Conference on P2P, Parallel, Grid, Cloud and Internet Computing*, pages 759–768. Springer, 2016.