Aerospace Systems
https://doi.org/10.1007/s42401-024-00279-0

REVIEW

Low-level memory and timing analysis of flight code for unmanned aerial systems

Micheal Saleab[1] · Franz Sax[1] · Johann Schumann[2] · Florian Holzapfel[1]

Received: 12 October 2023 / Revised: 2 January 2024 / Accepted: 31 January 2024

Abstract
The development of flight software for Unmanned Aerial Systems (UAS) is challenging due to the absence of an established development process defined by aerospace certification authorities. This research paper outlines our methods and tools for analyzing flight-critical UAS control software on the target hardware. We present our toolchain and methodology for evaluating the flight control computer stack, runtime memory, and timing characteristics. Additionally, we compare the performance of the flight control computer under various hardware and cache settings to justify, which hardware features should be enabled. The tools and processes employed in this research are deployable to any other development environment and are not restricted to the specific target hardware used in this paper.

Keywords Flight control computer · Safety-critical software · Memory safety · Timing analysis · Software certification

1 Introduction

This paper is based upon a conference paper [1], which has been extended and updated substantially.

All modern aircraft require on-board computing and software (SW) for their operation. However, the computer hardware (HW), the SW load, and the process how the SW is developed strongly depends on aircraft category and mission. For piloted commercial transports, safety is of utmost concern. The flight software, which supports the pilot, runs on large and redundant computer hardware. The safety-critical portions of the flight software must be developed according to the rigorous DO-178C [2] standard.

On the other hand, most Unmanned Aerial Vehicles (UAVs) or Unmanned Aerial Systems (UASs) are substantially smaller than transport aircraft and are remotely piloted or are flying autonomously. Often they are battery-powered. This imposes severe restrictions on the type of computing hardware and software to be used: power and weight limitations usually don't allow for redundant computers or sensor systems. Typically, small System-on-a-Chip (SoC) processors or microcontrollers are being used as UAS Flight Computers. On the software side, requirements and capabilities for SW running on-board as UAS are very different from commercial aircraft (AC):

- A UAS does not have any pilots on board. This means that much more automated functionality needs to be available on board. Even for a remote-piloted UAS, as a minimum, inner-loop control, navigation, and communication must be provided by the flight computer. Inherently unstable multi-copters require fast inner-loop attitude control.
- Functionality for Autonomous Operations (AO) requires an additional, often substantial software load to be executed on-board and in real-time, e.g., trajectory planning, on-board diagnostics, sensory and perception modules, and on-board decision-making.
- Because of limited computational capacities, often no or only small Operating Systems (OSes) are used. Typical examples include the ArduPilot [3] control software, μCOS-II [4], or the OSEK/AUTomotive Open System ARchitecture (AUTOSAR) [5]. Proper memory or tem-

✉ Micheal Saleab
micheal.saleab@tum.de

Franz Sax
franz.sax@tum.de

Johann Schumann
johann.m.schumann@nasa.gov

Florian Holzapfel
florian.holzapfel@tum.de

[1] Institute of Flight System Dynamics, Technische Universität München, Boltzmannstrasse 15, 85748 Garching, Germany

[2] KBR/Wyle LLC, NASA ARC, Moffett Field, CA 94035, USA

poral separation as required by the Aeronautical Radio, Incorporated (ARINC)-653 standard [6] or even features like virtual memory or Linux-style functionality is rarely found on UAS platforms.

- Safety risks for UAS operations are much lower than for commercial transports, Still a UAS can cause considerable harm to other aircraft (mid-air collisions) or human life on the ground. Therefore, the UAS flight SW must be designed and developed to high safety and quality standards.
  Unfortunately, there are currently no specific standards for UAS software systems. Standards like DO-178C [2] are "too heavyweight" and too expensive to implement. Software guidelines for autonomy software (e.g., the EASA Guidebook [7]) or standards for run-time assurance (e.g., ASTM F-3269 [8]) are still in their infancy.

Based upon the observation that all UAS software is based on low-level SW and OS components (e.g., interrupt routines, drivers, memory management) that are close to the hardware, this paper addresses the following research questions:

1. Which Verification and Validation (V&V) methods and state-of-the-art tools (open source when possible) can be used to improve and ensure the safety of low-level components of (advanced) UAS flight software?
2. Which important SW V&V issues, as mentioned, e.g., by DO-248 [9] (in particular non-determinism, memory/stack safety, timing, compiler optimization, and code coverage) can be addressed by a simple, mainly open-source toolchain and a light-weight development process for UAS flight software?

In this paper, we focus on the analysis of memory requirements and execution times under advanced hardware features, interrupt routines, and synchronization between different hardware components. Based upon guidance provided by the DO-178C standard and supplements, we present different techniques and measurements to support memory and timing safety for flight-critical UAS software. Modern flight computers consist of individual components that interact with each other. We present results of studies, on how such hardware component interaction can influence the overall behavior of the software system. An important component of analysis for memory safety is the analysis of the run-time stack. We discuss static and dynamic approaches for determining the worst-case stack usage.

The main focus of this paper is on the timing analysis of the flight software. Since compiler optimizations and hardware features of the Central Processing Unit (CPU) like caches or branch prediction heavily influence the run-time of the code, we first present measurements executed under different configurations. These provide insight into software and hardware configuration and yield a baseline for the Worst Case Execution Time (WCET) analysis.

In this paper, we present measurements showing the influence of cache usage and hardware optimizations on the run-time behavior on our hardware platform. We discuss the major approaches for estimating worst-case execution times with abstract interpretation, measuring high-water marks, and hybrid approaches. We present the results of experiments and measurements and discuss possible ways to reduce non-determinism in the software execution without sacrificing too much performance.

As our case study, we analyze the flight control software for an unmanned aircraft, which is developed at the Institute of Flight Systems Dynamics. The software for the Critical Flight Control Computer (CFCC) has been fully developed in a model-based manner with heavy use of automatic code generation. The target platform is a small, Power-PC MPC5777C-based computer system running a small operating system without memory management.

The results of these analyses provide numerous data points to support safety and reliability of the flight software. We will discuss, how these analyses could be combined with other verification and validation steps (static code analysis and coverage-based testing) to demonstrate the safety of our flight control system and how this process ultimately might support certification of UAS flight control software.

The rest of this paper is structured as follows: Sect. 2 provides the background about the UAS, the flight computer, and the flight control software for our case study. In Sect. 3 we discuss the requirements for verification and validation (V&V) as they are posed by relevant certification standards. Here, we focus on our topics of WCET analysis and memory-related issues. In Sect. 6.1 we describe the timing between the CPU and the Air Data and Attitude Heading Reference System (ADAHRS) and, moreover, present findings about synchronization issues. Section 5 focuses on the memory-related analyses for V&V: we present and compare techniques for stack analysis and discuss the effects of cache for timing analysis. In Sect. 6 we present techniques and experimental results of run-time measurements, which are the baseline for our study on Worst-Case Execution Time analysis, which is described in detail in Sect. 6.4. Based upon analysis techniques for memory and timing analysis, in Sect. 4, we propose a tool-chain that supports V&V of our model-based software development process and provides important information for future certification. Section 7 presents related work and Sect. 9 summarizes the paper and concludes.

2 Background

2.1 Embedded flight hardware

Despite their complexity, many UAS only have a relatively small and simple main flight computer. Figure 1A shows a typical configuration of a flight computer that is in charge of controlling the hovering engines, usually between 4 and 16 electrical motors that drive propellers.

The main flight computer, the Flight Control Computer (FCC) is a single-board micro-controller with Random Access Memory (RAM), flash RAM, and communication interfaces. Figure 1A shows an example board from our case study (see below). The FCC computer module receives inputs from the ADAHRS, provides attitude angles, rotational rates, and specific forces as well as airspeed and barometric altitude data. In many cases, an ARINC-429 protocol is used to transmit the data from the ADAHRS to the FCC. Additional input comes from the pilot operating the vehicle or the guidance and navigation computer. From these data, the necessary power/throttle settings for each of the motors is calculated and the necessary commands are sent to the Engine's Speed Controllers (ESCs) using a controller area network (CAN) bus interface.

The FCC board also contains power regulators, timers, Joint Test Action Group (JTAG) interfaces and other components, which are not of interest to our considerations.

Such a minimalistic hardware configuration can be typically found in small UAS; larger UAS will use redundant flight computers with higher complexity. Nevertheless, most of the issues discussed in this paper can be studied on the small hardware configuration.

2.2 Embedded flight software

As discussed above, the software running on the FCC is in charge of controlling the various motors of the UAS. In general, the SW structure for this task also applies to many other embedded systems. Data coming from sensors or other inputs need to be processed periodically, outputs computed and sent to output interfaces, and internal states need to be updated. In addition, other tasks for handling input/output (I/O) or for online hardware monitoring is also executed regularly.

On small embedded hardware platforms like those descri bed above, the software is executed under the control of a small real-time operating system (OS). Typical operating systems for such a platform include μCOS-II [4] or OSEK/AUTOSAR [5]. These provide substantially less functionality than Linux-style systems or VxWorks, but have a very small computational footprint.

Figure 2A shows a typical high-level software architecture. Sitting on top of the hardware, the OS provides functionality to run one or more software tasks in a (possibly preemptive) real-time manner. Interrupt service routines (ISRs) respond to hardware events, e.g., to read available data from a sensor interface. Driver components provide higher functionality for reading and processing sensor data and commands. Libraries, usually provided by the OS vendor, implement the necessary low-level functionality. Note that in many cases, such a system does not feature file systems, dynamic memory management, or interactive user I/O functionality (e.g., login, shell).

Therefore, the toolchain for development, debugging, testing, and deployment of safety-critical software on such computing platforms must be specifically tailored toward that hardware.

Figure 2B illustrates a typical way, how the embedded control software can be implemented. For each of the generic development steps, we discuss the tools that have been used in our case study. The algorithms and control logic for the flight software are developed using a Matlab/Simulink based tool-chain. For requirements management (functional requirements, like performance and operational behavior) the tool Polarion [10] was used. From the Simulink model, executable C code is generated using the Mathworks Embedded Coder. For building the full application code for the Critical Flight Controller (CFC), the Simulink generated C-Code, the OS code (μCOS-II-C-Code), and the driver code for the different interfaces are compiled and linked together with a version of the GCC (Gnu C Compiler) that uses the library "newlib" for the provision of, e.g., mathematical functions.

2.3 Our case study: electric vertical take-off and landing-transition vehicle

For our case study, we consider an Electric Vertical Takeoff (EVTOL) transition vehicle. The vehicle is controlled by several independent FCCs Flight Control Computers (FCCs) that are in charge of handling various subcomponents and functions. For this paper, we focus on the FCC for controlling the aircraft in the hover regime, called the "CFCC", which stands for "Critical Flight Control Computer".[1] In hover mode, the aircraft is actuated by 10 electric motors, whose propellers generate thrust in the downward direction, like ones used to see in commercial small drones. Via the ARINC-429 protocol, the FCC receives data from an ADAHRS that provides attitude angles, rotational rates, and specific forces, as well as airspeed and barometric altitude data. Further input comes from the pilot operating the vehicle (Fig. 1). From these data, the necessary speed for each of the 10 motors is calculated by an INDI (Incremental Nonlinear Dynamic Inversion)-controller and speed commands are

[1] For simplicity reasons, we assume that there is only one FCC, although the real UAS uses a redundant system architecture.

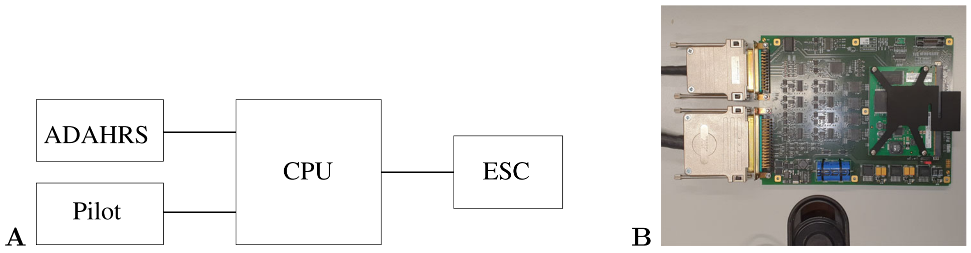

Fig. 1 **A** High-level architecture of the flight hardware, **B** image of our FCC target hardware

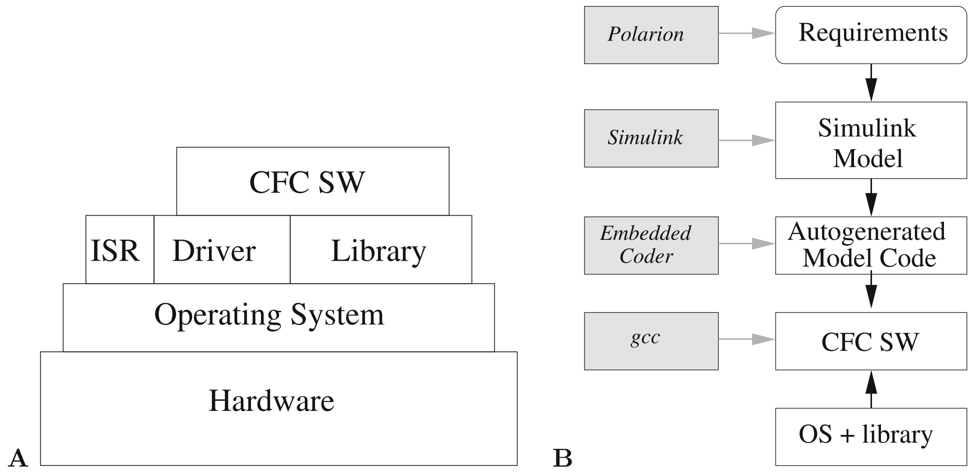

Fig. 2 **A** High-level architecture of the flight software. **B** Model-based software development process and tool-chain

sent to the respective Engine Speed Controllers (ESC) via a CAN bus interface.

The FCC itself consists of a Commercial off-the-shelf (COTS) Microsys miriac MPX-C5777 computing module that is built around an NXP MPC5777 micro-controller [11] with an e200z7 Performance Optimization With Enhanced RISC (PowerPC) running at 264 MHz. 256 KB of RAM is available for the controller application, and 4 MB of flash memory can be used for the application binary. CAN communication is established using the native CAN interfaces of the microcontroller, and the ARINC-429 communication is realized over a dedicated ARINC-429-interface chip connected via Serial Peripheral Interface (SPI). Of course, further interfaces are provided but are not relevant to our interests.

As operating system, a version of μCOS-II—a preemptive Real Time Operating System—that is tailored towards usage in safety-critical applications is used. Among other tasks that are related to I/O-handling and online hardware monitoring, the most important task for our investigations is the 'CFC_task'. In this task, CAN and ARINC-429 data from RAM buffers is transferred to the controller application 'CFC_step()', and after its execution, the computed actuator commands are written into the TX-buffers of the CAN-Interface from where they are then sent to the ESCs. Figure 2A shows the high-level software architecture.

The execution of this task typically fits into a periodic cycle time of 20 ms and there is still time for lower-priority tasks to execute until the cycle starts over. The CFC_task has the highest priority of all tasks being executed. Timing comes from a timer module that generates an interrupt every 1 ms and by that notifies the OS that a millisecond has passed. Periodic execution of the task is achieved by sending the CFC_task to sleep after going through the calculation of CFC_step() for an amount of time that results in its rescheduling 20 ms after its last scheduling. In the meantime, other tasks can be executed. As the CFC_task has the highest priority of all tasks, once its sleep time has passed, it immediately preempts the currently executing task and the cycle starts all over.

The controller algorithm itself, CFC_step() is developed with a Matlab/Simulink based tool-chain (Fig. 2B). For requirements management (functional requirements, like performance and operational behavior) Polarion [10] Via the tool SimPol [12] those high-level and low-level requirements are linked to the Simulink blocks of the controller model, which implement their required functionality. Using only a subset of the allowed Simulink functionality—which is checked via the Simulink Model Advisor—the C-code, generated from the model by Embedded Coder is well traceable to the Simulink model structure. For building the full application code for the CFC, the Simulink-generated C-Code, the μCOS-II-C-Code and the driver code for the different interfaces are compiled and linked together with a version of GNU Compiler Collection (GCC) that uses a version of newlib for the provision of, e.g., mathematical functions.

The control law design and its functional and performance testing are done within a Simulink environment, where a detailed flight dynamics model of the EVTOL is used as the plant. After all model-in-the-loop tests run successfully, the flight control algorithm is also tested on the target hardware. This is accomplished by the use of a Hardware-in-the-Loop (HIL) Setup. The HIL-Setup consists of a Real-Time-PC (RTPC) which contains and executes a model of every important component of the system architecture. This also includes the FCC. By virtually connecting the architecture components, a full virtual system simulation is achieved in software, comparable to the Simulink environment, but on a real-time PC. Pilot inputs can then be provided to the simulation by flight sticks from a cockpit mockup or a remote control and the vehicle position and attitude are forwarded to a visualization software such that a realistic flight simulator is obtained. The special hardware and software solution used in this HIL-simulation provides the possibility to substitute simulated entities with real entities by having the possibility to send out internal data via standard avionic interfaces, e.g., ARINC-429 or CAN. On the other side, CAN and ARINC-429 messages from outside can be received by the RTPC and be fed into the simulation. This allows connecting a real FCC with the controller executable on target, sending the ARINC-429 data from the simulation to the FCC, and forwarding the received ESC commands received via CAN to the ESC models. In the end, this gives the possibility to exercise the CFCC in a very realistic environment representative of its later application ("flying").

Of course, the FCC flight control algorithm can also be executed without any incoming sensor data ("idle"). In the paper, we will compare "idle" behavior with "flying" behavior to see, to which extent the behavior in the "flying" case can be deducted form the "idle" case—a setup that can be easily prepared in comparison to the time and resource-intensive setup of fully integrated hardware in the loop flight simulator.

3 Verification and validation

The intended application of the Friction Stir Welding (FSW) on small unmanned research aircraft (or optionally piloted aircraft) does not require the application of DO-178C for certification purposes. Yet, the standard offers important insight and enables us to design a software development and V&V process that could extended toward certification.

Therefore, for the analysis of our flight software, we align our V&V process and tools according to DO-178C [2] as well as the supplemental document DO-248.

In particular, results of software tests and analyses will be documented to supplement the "Software Reviews and Analysis" (see DO-178C Sec 6.3). The results of the timing analysis will be summarized and presented in the "Software Accomplishments Summary" (DO-178C, Sec 11.20.i). A detailed analysis will be carried out to support Objective 6 of DO-178C, Table A-5.

For the topics in this paper, we are specifically interested in guidance regarding non-determinism, memory, timing, and compiler optimizations. A detailed description of our tools and approaches will be presented in subsequent sections.

The RTCA FAQ document/supplement for DO-178C, named DO-248 [9], discusses topics including compiler optimization, code-inlining, and the use of data and instruction caches. FAQ#73 (3.73) is specifically concerned with timing measurements and worst-case timing. Section 3.73 lists several "factors, which can complicate the calculation of WCET". In the following, we list these factors and discuss, if they apply to our situation, and how we address those.
1. *Multiple interrupts that drive the system* This factor is not relevant in our case, because only one type of interrupt, namely the 1 ms heartbeat interrupt occurs in the system. 2. *Many decisions in the algorithm* The CFC, as generated from the Simulink model by Embedded Coder contains numerous conditional statements. The run-time behavior of the CFC code can be analyzed using WCET analysis taking into account all decisions inside the code, run-time measurements using a set of test cases that provide sufficient code coverage, or a hybrid testing and WCET analysis. Our approaches will be discussed in detail in Sect. 6.4. 3. *Use of processor cache for instructions or data* The use of data and instruction caches can substantially reduce the average execution time, but will also introduce non-determinism into the system which needs to be analyzed. 4. *Lack of real-time driven clock input* This factor is, in principle, of no concern in our architecture, where a hardware real-time clock drives and controls the execution. However, inaccuracies and drift of the hardware clocks in different components can add substantial problems (see Sect. 6.1). 5. *Differing methods of scheduling the tasks to be executed* In our architecture, priorities for the individual tasks are fixed. Furthermore, the central CFC task has, except for the error-handling task, which terminates the entire execution, the highest priority. So, the CFC task will never be preempted unless a system termination is required.
6. *Use of a real-time operating system* This architecture uses a small real-time operating system (see Sect. 2). However, due to simple scheduling, no dynamic generation/deletion of tasks, or complex kernel functions, WCET analysis is not influenced by these effects. Nevertheless, it is important to analyze the overall timing behavior of the real-time operating system and the connection between different hardware components (see Sect. 6.1).
7. *Operational behavior that is non-deterministic (for example stack usage)* The CFC software is fully deterministic. Since the CFC task is never invoked by an interrupt routine, stack sizes, and stack pointers (with respect to the task stack basis) are fully deterministic. The 1 ms heartbeat interrupt, which comes in asynchronously does not affect the stack of the CFC task, as the operating system provides a dedicated

interrupt stack. After execution of the ISR, the CFC stack will look exactly as before.

In the following sections, we will describe in detail some of the issues mentioned above, present methods and tools for analysis and mitigation. Based on that information, we propose a draft process and tool-chain to facilitate analysis and V&V of the CFC and UAS software in general.

4 Analysis tool-chain

Before we go into details of the individual methods and analysis tools, we will look at how they fit into the big picture of the software development and V&V process. Many processes and standards follow the traditional "V" shape, which defines stages of requirements, design, implementation, and then unit and system testing. Verification and validation activities are linking these various stages. Although the analysis methods discussed in this paper are very low-level and hardware-oriented, their use can influence early stages, as, for example, results of worst-case execution time analysis can have an impact on the design and even requirements.

Figure 3 shows our analysis and V&V tool-chain: in the middle column, we have the various phases and artifacts involved. These are managed and/or created by tools like Polarion (for requirements capture), Simulink [13], and Real-Time Workshop (RTW) for the automatic generation of major parts of the flight software. The C/C++ compiler, in our case GCC, and related tools generate the binary code for the execution on the target platform. Our analysis steps are shown in the right column. Whereas most of them are dealing with (generated) C/C++ code or even binary code, the analysis of the software design, in the form of Simulink models [13], can provide valuable results too. For this purpose, we are using the Simulink model advisor to inspect the Simulink models [14].

The Simulink coder is used to auto-generate the C-code, which implements the Simulink model functionality. Then, the auto-generated code is integrated into the CFCC low-level software, which includes: (a) board support package, (b) low-level drivers, (c) operating system, and (d) compiler libraries.

The entire software stack is analyzed using a static code analyzer. PolySpace [15] is a good candidate here since it is compatible and integrated into the MathWorks toolchain, which also includes Simulink, which was used to define and implement the controller (application layer).

Avstack [16] has been used to analyze the stack usage. By adding the "-fstack-usage" compiler flag as a compiler option, the GCC [17] compiler will auto-generate the maximum amount of stack used by each function in files with .su extension, this will be performed while generating the object files. These files are analyzable using the tool avstack, which generates a report with the stack usage for the entire application as an output. This tool is covered in more detail in Sect. 5.1.

Experiments were done to integrate a compiler built-in code coverage tool into our toolchain. Since we are using a GCC compiler, we used its coverage instrumentation feature, and the gcov [18] analysis tool. When an instruction of the instrumented code is executed, the corresponding coverage counter variable is incremented. After the program execution, a library function collects the information and writes it out to a file for use by an analysis and visualization tool. For use with our target hardware and operating system, we adapted that library to work in an embedded environment. It will be described in more detail in Sect. 5.2.

Lastly, we evaluated the worst-case execution time which is an important aspect for real-time systems. In addition, the WCET should be provided to the certification authority according to DO-178C [2]. We used a commercial product called RapiTime provided by Rapita. This tool instruments the software. Then, the instrumented software needs to be integrated into a framework that gathers the timing data from the instrumented software. The framework stores this data and provides it to the host PC which runs the RapiTime timing analysis. RapiTime generates a report, which contains the timing details that were computed from the data captured while running the instrumented executable on the target. The WCET analysis is shown in detail in Sect. 6.4.

In the following sections, we will describe the tools/methods used for low-level memory and timing analysis. We use our Case Study as a driving example and present the results of experiments.

5 Memory-related analysis

The flight code is usually implemented in C. There, we can distinguish between *static* memory, which is used to keep code, constants, and static variables, and *dynamic* memory, which is allocated and freed during the execution of the program. In-flight code developed in C, only the *stack* is considered dynamic memory. It is being used when functions are called, or when local variables are being used. At the end of the function call, the stack pointer is reset, thus destroying the local variables and freeing the memory. Explicit memory allocation with `malloc/free` or `new` is, due to the complexity of memory management and garbage collection, not allowed in flight code [2, 19].

Whereas it can be checked during compile-time if the static code and data fit into the given memory, the stack analysis is more complex and will be described in the following. Both static and dynamic memory are subject to hardware optimizations like enabling the caches or enabling the branch

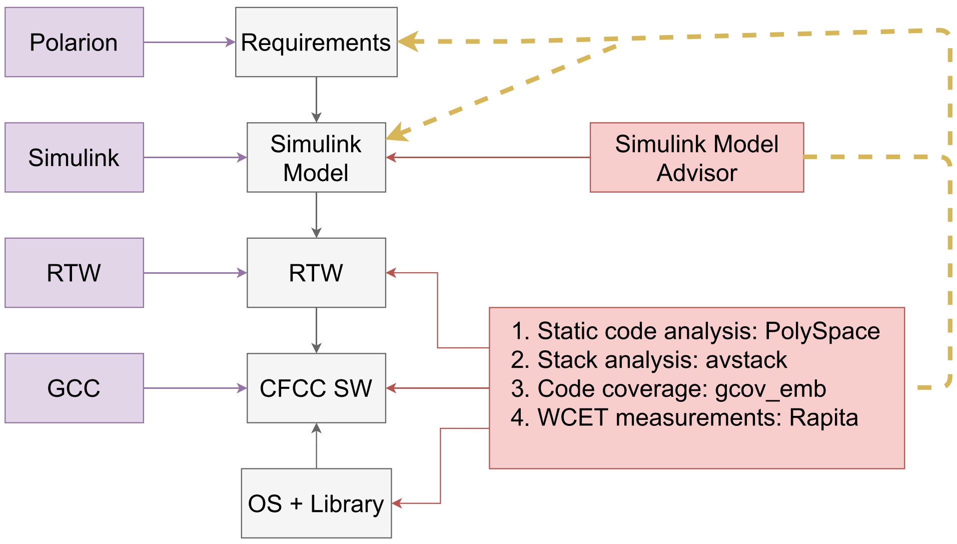
Polarion
Requirements
Simulink
Simulink Model
Simulink Model Advisor
RTW
RTW
1. Static code analysis: PolySpace
2. Stack analysis: avstack
3. Code coverage: gcov_emb
4. WCET measurements: Rapita
GCC
CFCC SW
OS + Library

Fig. 3 Our analysis and V&V tool-chain

prediction activation. Therefore, appropriate analyses need to be carried out (Sect. 6.2).

5.1 Stack analysis

Within the framework of μCOS-II, for every single task a constant amount of memory is reserved somewhere in RAM, where the task's private stack will be located. The size of those stack segments has to be statically set by the programmer in the source code before compile time. To gather information about the stack usage of the CFC_task—which is the most memory intensive task—a combination of GCC-functionality and the freely available Perl script "avstack.pl" [16] was used. Because GCC has to know how much stack space to reserve for every function when generating the machine code, this information can be readily obtained by adding the flag `-fstack-usage` to the compile command line options. With this flag for every source code file < `source>.c` a corresponding stack usage file < `source` > `.su` is created by the compiler that contains the size of memory used on the stack by the respective functions that are defined in the C-file. The script "avstack.pl" constructs a full call tree of all involved functions and adds up the contributions of the different functions, which are taken from the stack-usage files. By following the longest branch down the call tree, the theoretical maximum stack usage of the top-function can be calculated, if there are no recursion calls inside the software which is a hard requirement for our experiments.

To compare those numbers to the real stack usage on the target, a stack marking method was used: Before the first execution of the CFC_task, its stack is initialized such that every byte contains the value `0xA5`. As for the target hardware the stack grows from high to low addresses, this means that unused stack in the low-address-region will continue to contain the value `0xA5`, while high-address regions that were once written to typically not contain the value `0xA5` anymore. By implementing a routine that counts the number of `0xA5`-bytes from the lowest address of the stack space, the real stack usage can be determined after the execution of the code. Experiments in the "flying" setup show, that from the very first moment of stack checking after the first execution of the controller algorithm, the stack usage obtained by this method is constant. This is in line with expectations, as the controller is designed such that the same control law commands the airplane in its complete flight envelope and thus the code does not contain functions that are specific to specific flight maneuvers. Table 1 compares calculated and measured values for different compiler optimization settings.

Table 1 Stack usage computations and measurements

Optimization level	computed [Bytes]	measured [Bytes]
-O0	9120	9264
-O1	8776	8788
-O2	8760	8780
-O3	8768	8788

It can be seen that the order of magnitude is the same for all 4 cases, for the measured usage as well as the calculated usage. Further, measured and calculated usages are very close to each other for all different optimization levels, which tells us that the method works well. The measured stack usage is always a bit bigger than the computed stack usage, which is related to the fact that the operating system also reserves and initializes a small portion of the task stack when initializing—usage which is of course not taken into account by the GCC-based static stack analysis. Any interrupts that may enter the system at any time use a dedicated interrupt stack and by that do not influence these calculations and measurements.

5.2 Coverage analysis

An important validation task for any software is to check, which parts of the code have been exercised during the test runs. Although high code coverage does not necessarily reduce the number of possible software errors, it indicates that all necessary code has been touched by at least one test case. Obviously, code branches and loops can require multiple program executions to check that each branch (e.g., the 'then' branch and the 'else' branch is covered). Depending on the level of criticality, software standards can require different coverage criteria and metrics. For example, DO-178C [2], DAL-A requires code coverage and test coverage using Modified Condition/Decision Coverage (MC/DC). This metric does not only explore different branches of the code but also tests for coverage of complex conditional statements. In practice, full MC/DC coverage requires a substantial amount of test cases and is hard to achieve. For lower DAL levels, only statement and/or branch coverage is sufficient.

In general, code coverage is measured while the program is executed on test cases (or, rarely, during real operation). A multitude of tools, both COTS and open source, exist to facilitate this task. However, there are several issues to be considered: some tools perform instrumentation of the source code to acquire the necessary data during run-time, in such cases, the instrumented SW shall be part of the SW testing process, also the instrumentation shall be optimized to have a low impact on the system performance. For statement/branch coverage, most compilers, e.g., GCC, provide instrumentation on the machine code level, substantially cutting down on the required overhead. Also, the program under test should be exactly executed in the way it will be executed during the mission. This "fly as you test, test as you fly" principle [20] suggests that the software is designed and implemented in such a way that a code-coverage instrumentation (needed during testing) stays in the production build and does not violate any constraints. Therefore, a lightweight code coverage tool is to be preferred.

Secondly, the code coverage counters are stored in memory. However, read-out and export usually requires OS functions (like malloc, open, write, etc.), which are not necessarily available on an embedded system. For example, our case study system does not have a file system of malloc memory management.

To meet the requirements above, we chose to develop a customized tool using the code coverage instrumentation of GCC. Based upon an open-source implementation of coverage analysis [21], we extended the existing code base to (a) eliminate the use of file operations and memory allocation, and (b) to be able to extract the data using the gdb debugger on the hardware level. After extracting the data from the target hardware system, the coverage information can be analyzed and used to produce annotated code listings. Figure 4 shows screenshots of the visualization on various levels of detail. For the visualization, we used the open-source tool LCOV [22].

6 Timing-related analysis

6.1 Hardware and OS timing issues

For the stability of the controller algorithm, the signal delay in the closed control loop is an important parameter. The ADAHRS sends new data packets, containing the most recent data "taken from physics" every 20 ms. The CFC_task is also scheduled every 20 ms to read out ADAHRS data and compute updated motor commands. Now, the ADAHRS may send a new data packet to the CFCC, but the scheduling of the CFC algorithm relative to the ADAHRS is such that this data packet has to "wait" on the CFCC for, e.g., 15 ms until it can be used in the computation, there is a dedicated task for handling the processing of the ADAHRS data, and preparing the data for the controller, within this 15 ms. This task also handles the delivery of the data to the controller. These 15 ms are an effective delay in the control loop. Especially for INDI-based control in the innermost attitude loop, these delays should be minimized as much as possible as the phase margin here is typically small.

A further fact is that the 20 ms time difference between the sending of two subsequent data packets of the ADAHRS is based on the ADAHRS internal clock, while the scheduling of the CFC main algorithm every 20 ms is based on the CFCC internal clock. Of course, these clocks will have a certain drift relative to each other, which is, e.g., further dependent on the respective temperatures of their oscillators and their drift over time. This means that independent operation of those two devices, the timing relation between the sending of an ADAHRS data packet and its subsequent usage in the controller algorithm will change over time. In Fig. 5A, the time difference between the reception of the most recent ADAHRS data packet and the scheduling of the CFC_task are plotted over iterations of the controller. The data were obtained on the CFCC by subtracting two high-precision-timer values, one of the receptions of a certain ARINC-429 label that designates the start of a new ADAHRS data packet (which was recorded via a low-overhead interrupt) and the moment of scheduling the CFC_task. One can see, that this difference is not constant, but has a certain constant over time slope. After the difference reaches 20 ms, it flips over to zero, because the difference to the most recent reception of an ADAHRS signal is recorded. The irregular part in the middle comes from the behavior of the ADAHRS and introduces a jitter in the range of 2 ms. For V&V, the impact of this irregular and non-deterministic behavior on the overall software functionality must be studied and documented carefully.

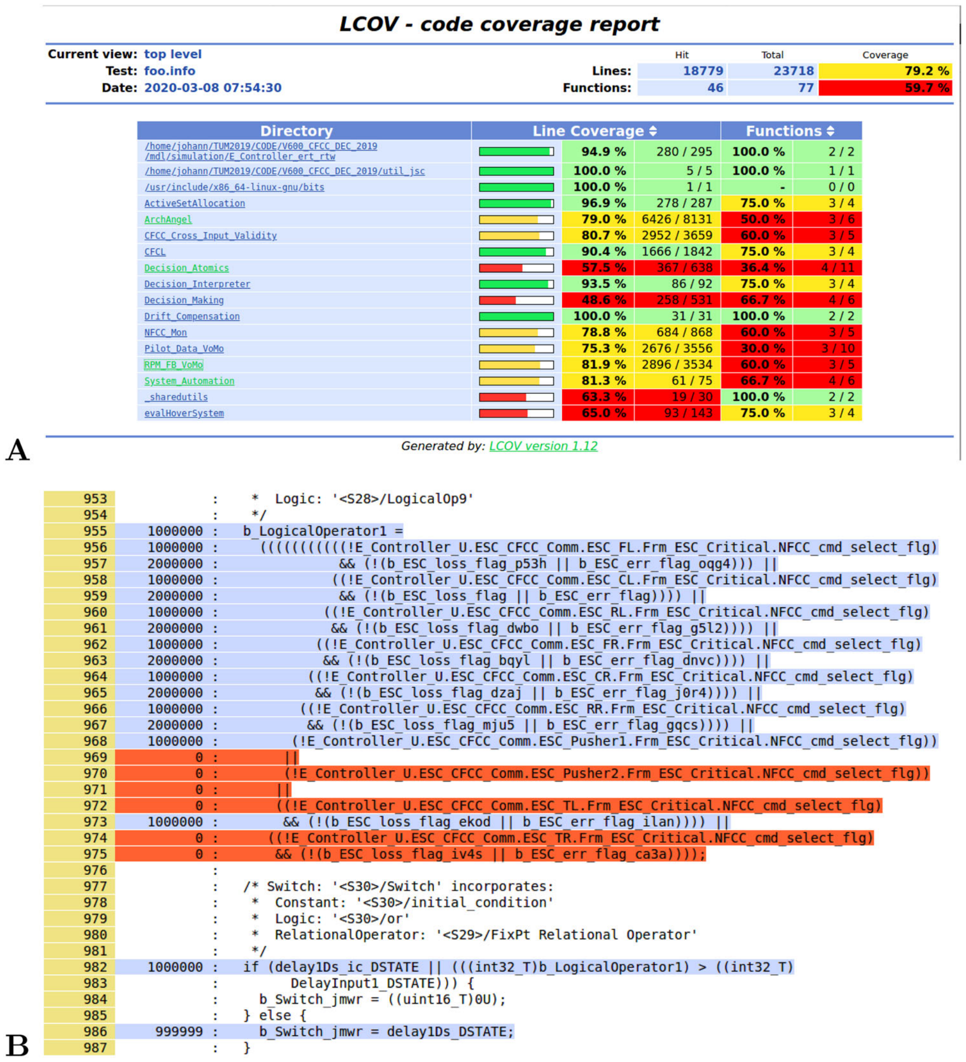

Fig. 4 **A** coverage summary after execution, **B** example for detailed analysis, indicating that some logical conditions have not been covered during a test (shown in red) (colour figure online)

LCOV - code coverage report

			Hit	Total	Coverage
Current view:	top level	Lines:	18779	23718	79.2 %
Test:	foo.info	Functions:	46	77	59.7 %
Date:	2020-03-08 07:54:30				

Directory	Line Coverage ⬍			Functions ⬍	
/home/johann/TUM2019/CODE/V600_CFCC_DEC_2019 /mdl/simulation/E_Controller_ert_rtw	🟩	94.9 %	280 / 295	100.0 %	2 / 2
/home/johann/TUM2019/CODE/V600_CFCC_DEC_2019/util_jsc	🟩	100.0 %	5 / 5	100.0 %	1 / 1
/usr/include/x86_64-linux-gnu/bits	🟩	100.0 %	1 / 1	-	0 / 0
ActiveSetAllocation	🟩	96.9 %	278 / 287	75.0 %	3 / 4
ArchAngel	🟨	79.0 %	6426 / 8131	50.0 %	3 / 6
CFCC_Cross_Input_Validity	🟨	80.7 %	2952 / 3659	60.0 %	3 / 5
CFCL	🟩	90.4 %	1666 / 1842	75.0 %	3 / 4
Decision_Atomics	🟥	57.5 %	367 / 638	36.4 %	4 / 11
Decision_Interpreter	🟩	93.5 %	86 / 92	75.0 %	3 / 4
Decision_Making	🟥	48.6 %	258 / 531	66.7 %	4 / 6
Drift_Compensation	🟩	100.0 %	31 / 31	100.0 %	2 / 2
NFCC_Mon	🟨	78.8 %	684 / 868	60.0 %	3 / 5
Pilot_Data_VoMo	🟨	75.3 %	2676 / 3556	30.0 %	3 / 10
RPM_FB_VoMo	🟨	81.9 %	2896 / 3534	60.0 %	3 / 5
System_Automation	🟨	81.3 %	61 / 75	66.7 %	4 / 6
_sharedutils	🟥	63.3 %	19 / 30	100.0 %	2 / 2
evalHoverSystem	🟥	65.0 %	93 / 143	75.0 %	3 / 4

Generated by: LCOV version 1.12

A

```
     953              :     *  Logic: '<S28>/LogicalOp9'
     954              :     */
     955      1000000 :   b_LogicalOperator1 =
     956      1000000 :     ((((((((((!E_Controller_U.ESC_CFCC_Comm.ESC_FL.Frm_ESC_Critical.NFCC_cmd_select_flg)
     957      2000000 :               && (!(b_ESC_loss_flag_p53h || b_ESC_err_flag_oqg4))) ||
     958      1000000 :              ((!E_Controller_U.ESC_CFCC_Comm.ESC_CL.Frm_ESC_Critical.NFCC_cmd_select_flg)
     959      2000000 :               && (!(b_ESC_loss_flag || b_ESC_err_flag)))) ||
     960      1000000 :             ((!E_Controller_U.ESC_CFCC_Comm.ESC_RL.Frm_ESC_Critical.NFCC_cmd_select_flg)
     961      2000000 :              && (!(b_ESC_loss_flag_dwbo || b_ESC_err_flag_g5l2)))) ||
     962      1000000 :            ((!E_Controller_U.ESC_CFCC_Comm.ESC_FR.Frm_ESC_Critical.NFCC_cmd_select_flg)
     963      2000000 :             && (!(b_ESC_loss_flag_bqyl || b_ESC_err_flag_dnvc)))) ||
     964      1000000 :           ((!E_Controller_U.ESC_CFCC_Comm.ESC_CR.Frm_ESC_Critical.NFCC_cmd_select_flg)
     965      2000000 :            && (!(b_ESC_loss_flag_dzaj || b_ESC_err_flag_j0r4)))) ||
     966      1000000 :          ((!E_Controller_U.ESC_CFCC_Comm.ESC_RR.Frm_ESC_Critical.NFCC_cmd_select_flg)
     967      2000000 :           && (!(b_ESC_loss_flag_mju5 || b_ESC_err_flag_gqcs)))) ||
     968      1000000 :         (!E_Controller_U.ESC_CFCC_Comm.ESC_Pusher1.Frm_ESC_Critical.NFCC_cmd_select_flg))
     969            0 :        ||
     970            0 :        (!E_Controller_U.ESC_CFCC_Comm.ESC_Pusher2.Frm_ESC_Critical.NFCC_cmd_select_flg))
     971            0 :       ||
     972            0 :       ((!E_Controller_U.ESC_CFCC_Comm.ESC_TL.Frm_ESC_Critical.NFCC_cmd_select_flg)
     973      1000000 :        && (!(b_ESC_loss_flag_ekod || b_ESC_err_flag_ilan)))) ||
     974            0 :      ((!E_Controller_U.ESC_CFCC_Comm.ESC_TR.Frm_ESC_Critical.NFCC_cmd_select_flg)
     975            0 :       && (!(b_ESC_loss_flag_iv4s || b_ESC_err_flag_ca3a))));
     976              :
     977              :   /* Switch: '<S30>/Switch' incorporates:
     978              :    *  Constant: '<S30>/initial_condition'
     979              :    *  Logic: '<S30>/or'
     980              :    *  RelationalOperator: '<S29>/FixPt Relational Operator'
     981              :    */
     982      1000000 :   if (delay1Ds_ic_DSTATE || (((int32_T)b_LogicalOperator1) > ((int32_T)
     983              :         DelayInput1_DSTATE))) {
     984              :     b_Switch_jmwr = ((uint16_T)0U);
     985              :   } else {
     986       999999 :     b_Switch_jmwr = delay1Ds_DSTATE;
     987              :   }
```

B

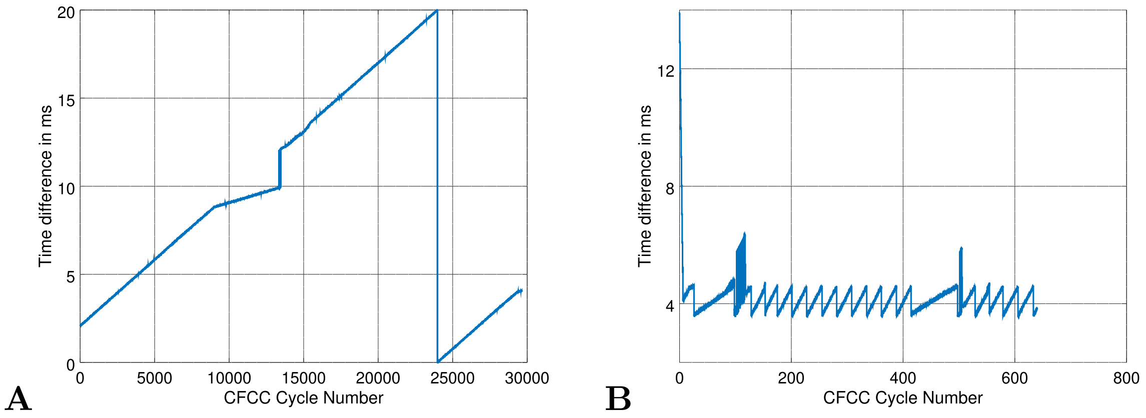

Fig. 5 Relative timing between **A** incoming ARINC-429 messages and control algorithm execution, **B** with execution synchronization mechanism

To account for those two shortcomings, namely the unnecessary introduction of a delay and the drifting of the offset between 0 and 20 ms, a continuous synchronization mechanism was implemented. Its goal is to start the execution of the controller 4 ms after receiving a new data packet, this 4 ms between the receiving of a new data packet and the controller execution is used to execute other tasks, which handle the data packet processing and pack the inputs needed for the controller execution. This synchronization mechanism is handled by a real-time operating system. At the end of controller execution, before the CFC_task is put to sleep for a certain amount of time, the times of the last reception of an ADAHRS packet and the time of the scheduling of the controller algorithm are compared. If in subsequent samples, this difference is too high or too low, a "re-synchronization millisecond" is either added or subtracted from the waiting time. By that, the CFCC "catches up" or "slows" down relative to the ADAHRS such that they reach and maintain a defined offset. Using an accordingly tuned algorithm, the synchronization also can handle the "jitter" phase of the ADAHRS transmission as can be seen in Fig. 5B. Now, a re-synchronization millisecond is added each time the offset grows too big, i.e., bigger than 4.5 ms in this case. Besides minimizing the delay between ADAHRS and CFCC, the delay is now also constant. This information helps the control law designers in their gain tuning and controller design.

6.2 Cache usage

To observe the effects of data and instruction caches on the execution time, a simple example (Listing 1) was put onto the target hardware. Here, an array x with 1000 integer elements has been initialized with some values. Then, inside the main loop, elements of the array x are accumulated such that each element x[i] is accessed twice. With a data cache of sufficient size, it can be expected that the value is stored in the cache with the first access; for the second access, the value stored in the cache can be accessed much faster [23].

Listing 1 Benchmark code for cache usage

```
uint32_t x[1000] = { 23, 4, 33, ... };
int cnt, acc, t1, t2, delta_t;
for(;;){
  t1 = STM_A_CNT;
  for(cnt = 0, acc = 0;cnt < 999; cnt++){
    acc += x[cnt]+x[cnt+1];
    }
  t2 = STM_A_CNT; delta_t = t2-t1; prt(delta_t);
}
```

For our analysis, the repeated execution time of the inner for-loop is measured in the following way: the MPC5777C contains a timer module called STM_A whose 32-bit timer—once started—increases at half of the processor frequency, i.e., 132 MHz. By noting down the timer values before and after the for-loop and sending the difference via a serial interface to a monitoring computer, execution time measurements can be taken in real-time and with minimal overhead. This is also the method employed in all further time measurements taken in this paper. The execution time measurement data are stored in a non-cached memory area, and the transmission to the monitoring computer is performed through a DMA channel. Transmission of the execution time measurement data is done after completing the required timing analysis, so as not to interrupt the timing analysis. One expects that there is a difference in execution times depending on the enabling of the data cache. With cache, the execution time should be smaller, as the processors L1-Cache with 16 KB is big enough to hold the whole array x inside it and no direct RAM accesses are needed, compared to the situation with disabled cache, where in every iteration some access to RAM takes place. However, for the first iteration with enabled cache, we expect a bigger run-time, as the array x is not yet in the cache, which highlights the benefits of enabling the cache, in case of cache-sensitive applications, and also confirms that in our fight controller application, the cache will become fully usable after the first iterations, which is accounted in the WCET analysis.

Table 2 Execution times in timer ticks for the code of listing 1

	1st	2nd	3rd	4th
Cache DIS	65,008	64,983	64,983	64,983
Cache EN	18,291	17,500	17,500	17,500

Running the experiment once with enabled and once with disabled caches showed (Table 2) that very quickly the execution times became stationary, and as expected, cache activation reduced the execution times, with the first execution taking longer than the following ones.

6.3 Execution time analysis

To get an overview of the typical execution times of the CFC_step() function, the same simple instrumentation as described above was used. The measurements were taken using different parameter settings: one parameter was the optimization level of the compiler (-O0, -O1, -O2, -O3), another one is the enabling of data and instruction cache, and the third one is the use of the branch prediction module of the processor. Further, measurements were taken in the "idle" setup and the "flying" setup using a pilot and the HIL-Flight Simulator.

A typical plot of the execution time over controller repetition number in the "flying" setup is shown in Fig. 6. One can see that in the beginning, the execution time is a bit smaller.

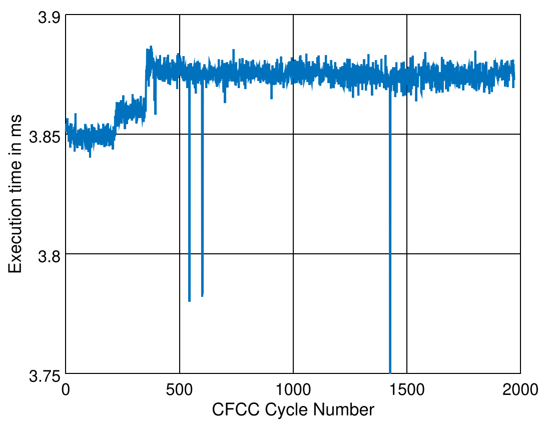

Fig. 6 Measured run-times of execution of the Controller algorithm in the "flying" scenario, Cache On, -O0, Branch prediction enabled

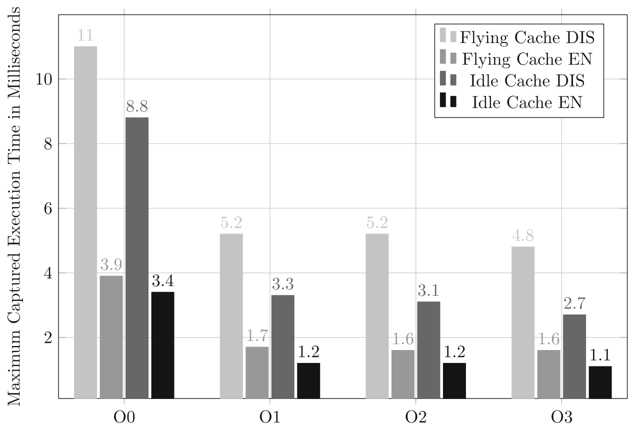

Fig. 7 Execution times of CFC_Step

This related to the fact, that the controller itself has to be armed and activated before being able to control the aircraft. The high plateau afterward corresponds to nominal operation with the CFCC in control. Characteristics are the sharp drops in the execution time that happen occasionally. This comes from the fact that in INDI, an iterative control allocation algorithm is used. In certain cases, this allocation converges very fast—e.g., in the cases, where there is no difference to the solution from the last execution. Nominal execution times are centered around a certain value, although subject to jitter that introduces a certain standard deviation. The bar plots in Fig. 7 document the respective numbers for the different settings.

These measurements reveal that activating the cache has a big impact as it always reduces the execution time by a factor of around 2.8. Compiler optimization also reduces execution times substantially. Here, the difference from O0 (no optimization) to O1 is the biggest, while the transitioning to O2 or O3 does not have a very big effect anymore. While enabling the O0 compiler flag, the GCC compiler the compiler reduces the execution time, and the code size and, while avoiding any optimizations that significantly affect the compilation time. By enabling O2 the GCC compiler performs all possible optimizations except space-speed trade-off, O2 has higher compilation time than O1. Lastly, by enabling O3 the GCC compiler enables all the optimizations flags. In general, execution times in the "flying" scenario are bigger than those in the "idle" scenario. Here, a factor of about 1.6 for disabled caches and a factor of 1.3 for enabled caches can be observed. These factors make it evident that timing measurements should not only be taken for the "idle" scenario but for a multitude of realistic flight scenarios. These measurements indicate that the maximum observed execution time of CFC_Step() is always within bounds here we run the flight control computer in a simulation environment while flying, to force the execution of our SW WCET software path. Even a build without optimization results in execution times well below the limit of 20 ms. However, we can not be sure that this big safety margin exists for all possible flight scenarios. Although the maximum observed execution time is always within bounds, WCET analysis is still needed to guarantee that during flight time, the flight control computer will always have the required idle time, and the execution of the CFC_Step will fulfill the timing requirement of the entire system.

6.4 Worst case execution time analysis

This section discusses the timing analysis and provides our approach to analyze the system's WCET in more detail. In real-time applications, timing information is important to understand the system behavior on the target hardware under all circumstances. In this section, we will discuss common tools and methodologies that are being used to analyze the worst-case execution time and present the results of measurements and timing analysis for our case study.

Besides the importance of analyzing the timing information to understand the application behavior, a WCET analysis is demanded by the aerospace certification authorities [Federal Aviation Administration (FAA) [24] and European Union Aviation Safety Agency (EASA) [25]]. Thus, it is good practice to integrate the needed tools and processes for analyzing the WCET into the tool-chain and the software development process.

Figure 8 from Rapita [26] shows how the execution times of a certain application are typically distributed. What is displayed here is a histogram of subsequent execution time measurements taken by running the application software on the target hardware.

Real-time systems typically come with an upper bound on the response time, which is the time passing between capturing the inputs, executing the functionality, and producing the output. The WCET of a function is the maximum interval of time this function can need to be executed on the compu-

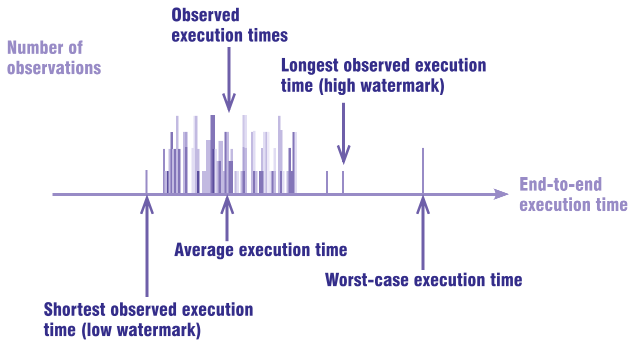
Number of observations
Observed execution times
Longest observed execution time (high watermark)
End-to-end execution time
Average execution time
Worst-case execution time
Shortest observed execution time (low watermark)

Fig. 8 Timing information observable during run-time. Figure reproduced with permission

tational platform. We did the WCET analysis to understand the maximum time frame demanded by the system, and to verify if it is acceptable or not. The WCET path is the path that results in having the longest execution time. The Low watermark of a certain functionality is the minimum observed execution time captured while executing this function on the target. Contrarily, the high watermark of a certain function is the maximum observed execution time captured while executing this function on the target. The WCET is always equal to or higher than the high watermark.

6.5 Tools and approaches for WCET analysis

Three well-known approaches are being used nowadays for analyzing/measuring the worst-case execution time:
Abstract Interpretation [27–29] is a static analysis method where the computations are abstracted and symbolically executed with respect to the hardware model, along with pipelines, caches, memory, and buses. This approach does not require real hardware as it relies on simulation. This will avoid having the target hardware as a blocking point. Abstract interpretation plays a great role at the start of new projects, where there could be a limitation in hardware, i.e. if the SW development started before having the actual hardware. Abstract interpretation requires comprehensive engineering work to implement models of the entire computational platform architecture, as well as the entire hardware configuration.
Analysis by measurements is based on running the software on the hardware and capturing timing information of the application and the software tasks, e.g., on the target hardware. The hardware is needed for this approach as the measurements will be performed on the target itself. If the data are captured on the target hardware itself, it needs to be transferred to the PC with the analysis tool using for example a debugger or a communication interface such as Universal Asynchronous Receiver–Transmitter (UART), CAN, or Ethernet.
Hybrid analysis [30] combines information from hardware timing measurements as well as from static code analysis.

6.6 Experiments and results

This section shows the WCET analysis using the tool RapiTime from Rapita, which is performing a hybrid WCET analysis. The controller software has been instrumented by RapiTime. The auto-generated instrumented software contains a certain ID for each code block, while executing a certain code block the instrumented software stores the timestamp captured at the start and end of executing this code block.[2] These timestamps get mapped to the code block ID assigned by the tool while generating the instrumented software. The tool uses these timestamps and the corresponding IDs to analyze the execution time for each code block separately. Each function consists of one or more code blocks, the function WCET is the sum of the WCET of each code block inside this function.

RapiTime provides several timing analysis profiles, we instrumented the software using the “Full Timing” profile; which instruments each code block; as explained above, to be able to get detailed timing information. Then, the instrumented software has been executed on the target hardware, the timing information has been collected while running the controller on the hardware. The target hardware sends the captured timing information to the host PC via a communication channel, in these experiments, we used the UART protocol to transmit the timing analysis data from the target hardware to the host PC. This timing information gets provided to RapiTime, the tool combines the captured timing information, additionally, it performs static code analysis on the source code. Lastly, the tool combines the captured run-time timing information and the data collected from the static code analysis to calculate the minimum execution time, average execution time, high watermark, and worst-case execution time.

We used RapiTime to estimate the WCET for four different executables, all executing the same main functionality (i.e., the CFC_task with the CFC_step() function being the most important ones), but with different settings in the startup phase of the processor, leading to certain features being used or not. The GCC compiler optimization was set to -O0 for all four code versions. All measurements were taken on target and in the “idle” scenario, by running the instrumented software on the target for a certain amount of time. These are the scenarios that we evaluated (a) Data and instruction caches ON, branch prediction ON, (b) Data and instruction caches ON, branch prediction ON, (c) Data and instruction

[2] Compilers assemblies statements and declarations presented in the source code into groups, each group is referring to a code block.

caches OFF, branch prediction ON, and (d) Data and instruction caches OFF, branch prediction OFF.

We afterward compared the timing analysis information generated by RapiTime for the four settings by having a look at the below timing matrices which are included in the RapiTime generated report:

- Minimum Execution Time: The minimal observed execution time for executing the controller software on the target.
- Average Execution Time: The average execution time observed while running the controller software on the target.
- High Water Mark: The maximum observed execution time while executing the controller software on the target. The WCET is always equal to or higher than the high watermark.
- Worst-Case Execution Time: RapiTime computes this number by assuming that each software block inside the application is being executed in its maximum observed execution time which has been observed while executing the application on the target. In addition, the tool assumes that each loop inside the application iterates the maximum observed number of iterations while running the application software on the target. RapiTime also assumes that for each loop the maximum execution time observed to execute a certain iteration will be the execution time needed for every single one of the iterations in that loop.

Figure 9 compares the minimum execution time, average execution time, high watermark, and worst-case execution time of each of the 4 code versions.

The analysis shows how enabling the caches and branch prediction impacts the four timing parameters.

- The WCET was reduced by 11.04% by enabling the branch prediction only.
- The WCET was reduced by 53.13% by enabling the instruction and data caches only.
- The WCET was reduced by 54.02% by enabling both the caches and branch prediction.
- The high-water mark was reduced by 18.72% by enabling the branch prediction only.
- The high-water mark was reduced by 60.29% by enabling the instruction and data caches only.
- The high-water mark was reduced by 62.54% by enabling both the caches and branch prediction.

It is underlined that the impact of branch prediction alone is not so significant, this concludes that the correctly predicted branches are not 100% in the WCET path. Furthermore, the minor improvement noticed in enabling branch prediction alone is related to our SW architecture and structure, as it does not include many branches. In contrast, to branch prediction, we noticed major improvements while enabling the caches ON alone, this is because our SW is cache-sensitive, due to the enormous amount of lookup tables which is used by the flight controller.

The results show that enabling the caches and branch prediction reduced the controller execution time on the target hardware. For any controller benchmark, this timing analysis can be performed to examine the impact of any hardware feature on the average execution time and WCET. Developers can select the right hardware features to be enabled, by comparing the WCET for different hardware configurations.

7 Related work

In the research paper “Execution time analysis and optimization techniques in the model-based development of a flight control software” [32], they approached enhancing the execution time using a formally verified compiler and enabling the compiler optimization. CompCert from Absint is used as a compiler and has been evaluated in this case study. The flight management controller has been developed using model-based development, and the software has been auto-generated from the Simulink model. This current case study is performed on a flight control application; in which safety perspectives were considered. The enhancement is assessed by evaluating the execution time measurements on the hardware using a real-time environment and a static worst-case execution time analysis using aiT from Absint.

This paper concluded that it is conceivable to get a safe execution time bound for Simulink model-based developed software. To get constrained limits for the investigated WCET without exerting a tremendous amount of effort in the analysis. The use of the CompCert [33] prompts an enormous improvement in the execution time in contrast with the non-optimized compilation. CompCert utilization improved the execution time with a margin more significant than the improvements achieved by optimizing the auto-generated code. they recommended applying both CompCert optimizations in combination with auto code generation optimization to achieve the maximum execution time enhancement.

In the research “Impact of Cache Partitioning on Multi-Tasking Real Time Embedded Systems” [34], the authors discussed cache partitioning and its impact on multi-tasking real-time applications. Cache partitioning approaches are extensively used as an interpretation for the cache interference quandary. Their main concern is that real-time embedded systems need to limit task real-time usage, rather than just improving the cache hits/misses rates, by reducing the cache misses ratio.

They presented the partitioning issue in the form of an optimization problem. The optimization problem is in charge

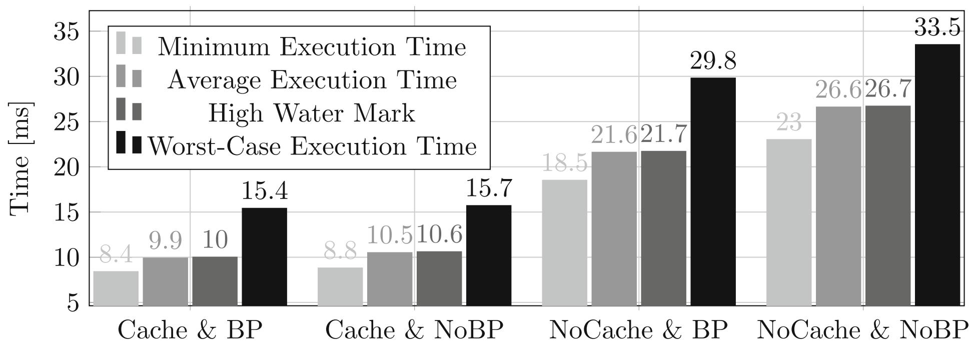

Fig. 9 Comparison of timing analyses carried out by RapiTime [31], BP refers to branch prediction

of handling the size of cache partitions, in addition, to assigning the partitions to the real-time tasks. They aim to improve the task's worst-scenario utilization by resolving this problem. They presented the proposed algorithm in their research paper. Their results show that the cache partitioning problem can lead to: (a) increasing real-time schedulability, and (b) improving the task run-time utilization by 15%. It could be beneficial to combine the tool-chain and process mentioned in our case study, with cache partitioning techniques, divide the controller application, into several tasks, see the impact of the proposed approach in the worst-case execution time, and compare the timing measurements with the scenario of just enabling the cache without the cache partitioning.

The effective use of multi-core platforms is one of the prime challenges in aerospace. The introduction of multi-core platforms is especially challenging for the high criticality levels of Design Assurance Level A (DAL-A) and Design Assurance Level B (DAL-B). Multi-core platforms usage in aerospace applications is defined in Certification Authorities Software Team (CAST)-32A [35] as they showed the challenges of the multi-core platforms. The paper "Multicore In Real-Time Systems - Temporal Isolation Challenges Due To Shared Resources" [36]; introduces multi-core temporal isolation techniques to allow the usage of advanced computational platforms in aerospace applications by applying the following temporal isolation: (a) system bus, (b) memory, (c) cache, (d) logical units, and (e) pipeline stages.

Applying the analysis measurements that have been discussed in our case study, especially the WCET analysis on a multi-core platform is a promising topic for the future of aerospace. Since the silicon industry is moving very fast, and nowadays there are advanced computational platforms available. However, this leads to making the analyses more complicated, it's more challenging to analyze the WCET for applications running on these platforms, due to the interaction between multi-core and shared resources. Also, the new platforms provide more advanced hardware features, which results in more complicated hardware architecture; this makes the WCET analysis more challenging, and certifying the hardware according to DO-254 [37] is not an easy task.

For measuring code coverage, there are numerous COTS and Open Source tools available. For an overview see, e.g., [38].

8 Recommendations

In this section, we state our recommendations which shall ease the development strategy and flight systems certification process. We found out that planning the tools which shall be used in code verification and providing the DO-178 certification artifacts, need to be done at an early stage in the development cycle, we have seen also that even for low-criticality level software projects tools like Polyspace for static code analysis, avstack for stack analysis, gcov for code coverage, and RapiTime for WCET measurements, can be very useful to avoid SW issues, and errors at late stages in the projects. These verification tools can run automatically through CI, which will reduce the overhead of running the verification cycles at an early stage in the project.

In addition to tooling planning, V&V planning plays a role as well, in aerospace software projects, the relation between V&V tasks, and certification requirements needs to be highlighted at an early stage in the projects, and the definition of the V&V tasks needs to be aligned with the certification requirement, in this approach the V&V itself shall provide the certification artifacts during the development cycle. This will make the certification process smoother and more convenient.

Moreover, defining test cases, which test the system against its high-level and low-level requirements, can ease the V&V right side tasks, since testing tasks consume most of the project development time, it makes sense to start testing in an early phase, and run the test cases on the developed functions using modern Continuous Integration (CI) approaches.

We found that early software and resources analysis provides, valuable information for the entire project, for example, a high worst-case execution time for some controller functions, can be detected if each developed controller module has its worst-case execution time analysis once it's fully developed, before integrating the full flight controller modules. This can highlight if a certain module has a high worst-case execution time margin and needs further optimization, or even needs changes in its design, or hardware configuration needs changes, for example, cache configuration, branch prediction, or in the worst scenario, the computing platform needs to be changed as to provide sufficient computing resources to fulfill the execution of the flight controller algorithms with the required idle time needed by the certification authorities, same goes for other analysis as stack analysis, and static code analysis, this analysis helps in avoiding major project issues in the late development stage.

While doing intensive SW analysis and testing during the V&V cycles, many tools need to instrument the software, for providing testing and analysis data, for example, run-time test cases, test coverage, or worst-case execution time. This results in providing the analysis on a different code version than the developed initial code. In this case, analysis and software verification verified the instrumented software and not the initially developed software, its recommended to fly with the instrumented software; as moving the instrumented code, needs to prove, that the instrumented software has the same analysis and verification aspects as the instrumented software. However, the instrumented software uses additional memory resources, and CPU load, which needs to be reserved for it during the project resources planning.

Many projects only perform static and run-time analysis on the application level and avoid examining the operating system and low-level artifacts as a micro-controller abstraction layer. However, the interaction of such low-level artifacts and the exact SoC used for the project can cause HW/SW failures, It is recommended to include the operating system and the micro-controller abstraction layer SW in the validation and verification process, to examine if the entire system is working as expected and the operating system, provides the required functionalities on the selected SoC, also timing dependencies between an operating system and application need to be examined since the operating system adds additional latency for providing some functions which are used by the flight controller, this latency needs to be considered in the analysis. In addition, to this, the operating system and the micro-controller abstraction layer, reserve memory resources, and also share the cache lines with the flight controller, these resource sharing and conflicts need to be considered in the HW/SW analysis.

Hardware configuration plays a significant role in the entire SW analysis, for example, cache and branch prediction are significant topics in flight computer development, because flight algorithms are cache-sensitive, and also use many branch scenarios. We found that activating the cache reduces the execution time by a factor of 2.8, and the controller execution time is higher during the flight time, with a factor of 1.6 in case of cache is disabled, and a factor of 1.3 if the cache is enabled. This shows how the flight algorithms are cache-sensitive. This opens the topic of cache optimization to extend the benefits gained from enabling the cache. For example, cache lock-down and cache coloring can help significantly improve system determinism. In addition, to the hardware configuration and its impact on the entire flight computer performance, compiler settings as the optimization level, can highly influence the flight controller memory and CPU load, it's recommended that every flight controller project study the available compiler options, and its impact on the project. In our project, we found the execution time reduction gained by -O1 instead of -O0 is significant. However, moving to -O2 or -O3 did not result in further significant optimizations. In conclusion in general HW configuration and compiler settings can result in improving the execution time by a whole order of magnitude in corresponding scenarios.

9 Conclusions

In this section, we conclude our research; which has been represented in our study case. Integration of worst-case execution time analysis tools to the tool-chain is a good practice especially for aerospace applications since it allows early understanding of the WCET and the percentage of which the WCET is growing over the development of new controller (application) features. in addition, running the application on the host PC processor architecture is not enough to understand how the application runs and behaves on the target hardware. Also, we managed to compare the margin of growth between the average execution time, high watermark execution time while running the full software stack on the target hardware, and the theoretical analysis of the WCET. This allows a better understanding of the application and the hardware behavior.

The utilization of the avstack tool was straightforward, the tool uses auto-generated stack information delivered by the GCC compiler using the compiler "-fstack-usage" flag. The tool uses the auto-generated stack information as input and generates the maximum stack size utilized for each function. This is a good practice to be integrated into the tool-chain, for a better understanding of the stack usage.

As shown in the related work several approaches could be added to the systems from similar study cases; to improve the run-time performance. Replacing the GCC with CompCert from Absint could allow us to enable the compiler optimizer while having a verified executable, this could result in a lower WCET. Besides, it's fascinating to try applying our toolchain on a multi-core platform and study how the WCET analysis can be integrated into the toolchain and process in the case of multi-core platforms.

Funding Open Access funding enabled and organized by Projekt DEAL. This research is funded by the Bavarian Ministry of Economic Affairs, Energy and Technology. Johann Schumann contributed to this

project while at the Institute of Flight Dynamics on leave from KBR in early 2020.

Declarations

Conflict of interest The authors have not disclosed any competing interests.

References

1. Saleab M, Sax F, Schumann J, Holzapfel F (2021) Toward timing and memory analysis for UAS flight code. In: AIAA Scitech 2021 Forum, p 1918. AIAA. https://doi.org/10.2514/6.2021-1918
2. Radio Technical Commission for Aeronautics (RTCA) (2012) Incorporated and The European Organisation for Civil Aviation Equipment (EUROCAE): DO-178C: Software considerations in airborne systems and equipment certification
3. Ardupilot.com: APM:Plane, Open Source Fixed-wing Aircraft UAV. http://plane.ardupilot.com
4. Labrosse JJ (2002) μC/OS the real-time kernel, 2nd Ed. CRC Press, Boca Raton. https://doi.org/10.1201/9781482295603
5. Consortium A (2014) Automotive open system architecture (AUTOSAR) Revision 4.2.1. AUTOSAR
6. Dubey A, Karsai G, Kereskenyi R, Mahadevan M (2010) A real-time component framework: experience with CCM and ARINC-653. In: IEEE international symposium on object-oriented real-time distributed computing
7. EASA (2021) EASA concept paper: first usable guidance for level 1 machine learning applications. Technical report, European Aviation Safety Agency
8. Nagarajan P, Kannan S.K, Torens C, Vukas M.E, Wilber GF (2021) ASTM F3269—an industry standard on run time assurance for aircraft systems. https://doi.org/10.2514/6.2021-0525
9. Radio Technical Commission for Aeronautics (RTCA) (2011) DO-248C, supporting information for DO-178C and DO-278A
10. Polarion, Siemens PLM Software https://polarion.plm.automation.siemens.com. Accessed 19 Feb 2023
11. NXP MPC5777C https://www.nxp.com/products/processors-and-microcontrollers/power-architecture/mpc55xx-5xxx-mcus/ultra-reliable-mpc57xx-mcus/ultra-reliable-mpc5777c-mcu-for-automotive-and-industrial-engine-management:MPC5777C. Accessed 19 Feb 2023
12. SimPol, TUM-FSD https://www.fsd.lrg.tum.de/software/simpol. Accessed 19 Feb 2023
13. Simulink, MathWorks https://de.mathworks.com/products/simulink.html. Accessed 19 Feb 2023
14. Simulink Model Advisor, MathWorks https://de.mathworks.com/help/simulink/ug/select-and-run-model-advisor-checks.html. Accessed 19 Feb 2023
15. Polyspace, MathWorks https://de.mathworks.com/products/polyspace.html. Accessed 19 Feb 2023
16. Daniel Beer, avstack.pl https://dlbeer.co.nz/oss/avstack.html. Accessed 19 Feb 2023
17. Stallman RM (2003) GCC Developer Community: using the gnu compiler collection for gcc version 4.9.4
18. Gcov using the GNU https://gcc.gnu.org/onlinedocs/gcc/Gcov.html. Accessed 19 Feb 2023
19. Graves S (2011) Justifiably taboo: avoiding malloc-free apis in military/aerospace embedded code. Military Embedded Systems
20. NASA lessons learned https://llis.nasa.gov/lesson/1196. Accessed 19 Feb 2023
21. libgcov-embedded https://github.com/reeteshranjan/libgcov-embedded. Accessed 19 Feb 2023
22. LCOV Coverage Tool https://github.com/linux-test-project/lcov. Accessed 25 Mar 2023
23. Boncz P (2009) Processor cache. In: Encyclopedia of database systems. Springer, Boston, pp 2195–2196. https://doi.org/10.1007/978-0-387-39940-9684
24. Federal Aviation Administration https://www.faa.gov. Accessed 19 Feb 2023
25. European Aviation Safety Agency https://www.easa.europa.eu. Accessed 19 Feb 2023
26. Rapita Systems https://www.rapitasystems.com. Accessed 19 Feb 2023
27. aiT, Absint https://www.absint.com/ait. Accessed 19 Feb 2023
28. Ferdinand C, Heckmann R, Wilhelm R (2006) Analyzing the worst-case execution time by abstract interpretation of executable code. In: Automotive software—connected services in mobile networks. lecture notes in computer science, vol 4147. Springer, San Diego, pp 1–14. https://doi.org/10.1007/11823063_1
29. Byhlin S, Ermedahl A, Gustafsson J, Lisper B (2005) Applying static wcet analysis to automotive communication software. In: 17th Euromicro conference on real-time systems (ECRTS'05), pp 249–258 . https://doi.org/10.1109/ECRTS.2005.7
30. Betts A, Merriam N, Bernat G (2010) Hybrid measurement-based WCET analysis at the source level using object-level traces. In: Lisper B (ed) 10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010). OpenAccess Series in Informatics (OASIcs), vol 15. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, pp 54–63. https://doi.org/10.4230/OASIcs.WCET.2010.54 . The printed version of the WCET'10 proceedings are published by OCG (http://www.ocg.at). http://drops.dagstuhl.de/opus/volltexte/2010/2825
31. RapiTime, Rapita Systems https://www.rapitasystems.com/products/rapitime/. Accessed 19 Feb 2023
32. Nurnberger K, Hochstrasser M, Holzapfel F (2017) Execution time analysis and optimisation techniques in the model-based development of a flight control software. Inst Eng Technol (IET) Cyber-Phys Syst Theory Appl 2:57–64. https://doi.org/10.1049/iet-cps.2016.0046
33. CompCert, Absint https://www.absint.com/compcert/index.html/. Accessed 19 Feb 2023
34. Bui B, Caccamo M, Sha L, Martinez J (2008) Impact of cache partitioning on multi-tasking real time embedded systems. In: 2008 14th Institute of electrical and electronics engineers (IEEE) international conference on embedded and real-time computing systems and applications, pp 101–110. https://doi.org/10.1109/RTCSA.2008.42
35. Certification Authorities Software Team (CAST) (2016) Position paper CAST-32A (multi-core processors)
36. Kotaba O, Nowotsch J, Paulitsch M, Petters S.M, Theiling H (2013) Multicore in real-time systems—temporal isolation challenges due to shared resources. In: 16th Design, automation & test in Europe conference and exhibition (DATE)

37. Radio Technical Commission for Aeronautics (RTCA) (2000) Incorporated and The European Organisation for Civil Aviation Equipment (EUROCAE): DO-254: design assurance guidance for airborne electronic hardware
38. Open source code coverage tools https://stackify.com/code-coverage-tools/. Accessed 19 Feb 2023