UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

BRUNO BOHRER COZER

Evaluation of FACE Implementation on Real-Time Avionics Performance

Thesis presented in partial fulfillment
of the requirements for the degree of
Master of Computer Science.

Advisor: Prof. Dr. Edison Pignaton de Freitas

Porto Alegre
2019

CIP – CATALOGUING-IN-PUBLICATION

Cozer, Bruno Bohrer

Evaluation of FACE Implementation on Real-Time Avionics Performance / Bruno Bohrer Cozer. – Porto Alegre: PPGC da UFRGS, 2019.

52 f.:il.

Orientador: Edison Pignaton de Freitas.

Thesis (Master) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação. Porto Alegre, BR – RS, 2019. Advisor: Edison Pignaton de Freitas.

1. Real-time Systems. 2. Embedded Systems 3. Avionics. 4. Performance. 5. Systems Architecture I. Freitas, Edison Pignaton de. II. Título.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Rui Vicente Oppermann
Vice-Reitora: Prof[a]. Jane Fraga Tutikian
Pró-Reitor de Pós-Graduação: Prof. Celso Giannetti Loureiro Chaves
Diretora do Instituto de Informática: Prof[a]. Carla Maria Dal Sasso Freitas
Coordenadora do PPGC: Prof[a]. Luciana Salete Buriol
Bibliotecária-Chefe do Instituto de Informática: Beatriz Regina Bastos Haro

You have a choice: you can either create your own future, or you can become the victim of a future that someone else creates for you. By seizing the transformation opportunities, you are seizing the opportunity to create your own future.

Vice Admiral (ret.) Arthur K. Cebrowski, US Navy.

AGRADECIMENTOS

Primeiramente, agradeço ao meu orientador, Prof. Dr. Edison Pignaton de Freitas, pelo incansável incentivo, otimismo e orientação que fizeram este trabalho chegar até aqui. Agradeço também à colega e amiga Ana Cláudia de Almeida Bordignon, cuja participação foi fundamental para a conclusão deste trabalho.

Agradeço ainda à minha esposa Dra. Aline da Silva Gonçalves Cozer pelo suporte acadêmico e emocional, fundamental nas horas mais difíceis. Dedico este trabalho a ela, a nosso filho Bento, e a meu pai, Evandro Luiz Cozer, com quem aprendi que a educação é o bem mais valioso que podemos obter.

ABSTRACT

Although systems architectures like IMA allowed avionics systems to became more modular and concentrated, saving space, power and weight on aircraft, there was still issues with reuse, modularity and vendor lock to be enhanced on mission systems. Open Systems Architectures (OSA) have been developed and explored by avionics developers and contractors in order to benefit from easier reuse, reduced costs and shorter time-to-field thanks to the modularity and common interfaces brought by such architectures. FACE is growing example of OSA framework defined to achieve these goals for mission software applications through the use of standardized abstraction layers. However, the insertion of such layers may impose overheads on performance that could jeopardize systems key features, especially on hardware platforms with strict space and power constraints.

This work discusses this tradeoff, comparing it with similar scenarios in different areas. Through the implementation of an experiment comparing five different performance metrics implemented on two sets of application versions: with and without FACE adherence, using two different hardware platforms, it was possible to quantify performance impacts brought by the OSA adoption. The obtained results showed that there may be impacts depending on computation type performed by the mission application. However, in the majority of cases the quantified overhead was less than 5% when comparing the same performance metric in a non-OSA implementation, therefore not risking the benefits brought by OSA modularity.

Keywords: Real-time Systems. Embedded Systems. Avionics. Performance. Systems Architecture.

Avaliação da Implementação de FACE no desempenho de Sistemas Aviônicos de Tempo Real

RESUMO

Apesar de arquiteturas de sistema com IMA terem permitido os sistemas aviônicos se tornarem mais modulares e concentrados, economizando espaço, peso e consumo de energia em aeronaves, ainda havia problemas com reuso, modularidade e falta de fornecedores alternativos que necessitavam serem melhoradas em sistemas de missão. Arquiteturas Abertas de Sistema (OSA) tem sido desenvolvidas e exploradas por fornecedores de aviônicos e contratantes de forma a se beneficiar do reuso mais fácil, custos reduzidos e menor tempo de desenvolvimento graças à modularidade e uso de interfaces em comum trazido por este tipo de arquiteturas. FACE é um exemplo crescente de ambiente OSA definido para alcançar estes objetivos para aplicações de software de missão através do uso de camadas de abstração padrão. Entretanto, a inserção destas camadas pode impôr acréscimos na performance que poderiam pôr em risco características chave dos sistemas, especialmente em plataformas de hardware com restrições de espaço e energia.

Este trabalho discute esta dicotomia, comparando com cenários similares em outras áreas. Através da implementação de um experimento comparando cinco métricas de performance diferentes implementadas em dois conjuntos de versões de aplicação: com e sem aderência a FACE, usando duas plataformas de hardware diferentes, foi possível quantificar os impactos de performance trazidos pela adoção de OSA. Os resultados obtidos mostram que pode haver impactos dependendo do tipo de computação realizado pela aplicação de missão. Entretanto, na maioria dos casos o impacto quantificado foi inferior a 5% quando comparado à mesma métrica de performance em uma implementação sem OSA, desta forma não arriscando os benefícios trazidos pela modularidade de OSA.

Palavras-chave: Sistemas de tempo real. Sistemas Embarcados. Sistemas Aviônicos. Desempenho. Arquitetura de Sistemas.

LIST OF FIGURES

Figure 2.1 – One Possible Way of Applying MIL-STD-498 to Program Strategy 17
Figure 2.2 – Relationship between DO-178B Certification Artifacts 19
Figure 2.3 – Federated vs. IMA Architectures 21
Figure 2.4 – ARINC 653 Software Architecture Example 22
Figure 2.5– Common Operational Environment Examples 24
Figure 2.6 – FACE Architectural Segments Example 26
Figure 3.1 – General workflow of OCLoptimizer 31
Figure 3.2 – CAS product architecture for FACE 33
Figure 4.1 – AIL structure 36
Figure 4.2 – AIL FACE Architecture 37
Figure 4.3 – AIL non-FACE (flat) architecture 38
Figure 4.6 – x86 Platform Setup (illustrative) 44
Figure 4.7 – Raspberry Pi 3 Platform Setup (illustrative) 45
Figure 5.1 – Task Clock Time 47
Figure 5.2 – Task Executed Cycles 50
Figure 5.3 – Executed Instructions 52
Figure 5.4 – Cache References 54
Figure 5.5 – Cache Misses 56

LIST OF TABLES

Table 2.1 – Failure Types and Consequences 18
Table 2.2 – Design Assurance Levels per Failure Types. 18
Table 2.3 – Available POSIX methods per FACE profile. 29
Table 3.1 – Scope comparison between related works and this work. 34
Table 5.1 – Complete Statistics for Task Clock Time. 49
Table 5.2 – Complete Statistics for Executed Cycles 51
Table 5.3 – Complete Statistics for Executed Instructions........ 53
Table 5.4 – Complete Statistics for Cache References. 55
Table 5.5 – Complete Statistics for Cache Misses. 57

LIST OF LISTINGS

Listing 4.1 – FACE PCS Segment Source Code 39
Listing 4.2 – FACE TSS Segment Source Code 39
Listing 4.3 – FACE PSSS Segment Source Code 40
Listing 4.4 – FACE IOSS Segment Source Code 40
Listing 4.5 – FACE OSS Segment Source Code 41
Listing 4.6 – Non-FACE Equivalent Source Code 42

LIST OF ABBREVIATIONS AND ACRONYMS

AIL	Avionics Infrastructure Library
API	Application Programming Interface
ARINC	Aeronautical Radio, Inc.
CA	Certification Authority
CAN	Controlled Area Network
CAS	Collision Avoidance System
COE	Common Operational Environment
COTS	Commercial Off-the-shelf
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
DAL	Design Assurance Level
DME	Distance Measuring Equipment
DIJ	Dijsktra Algorithm
EUROCAE	European Organization for Civil Aviation Equipment
FACE	Future Airborne Capability Environment
FACE-TS	Future Airborne Capability Environment Technical Standard
FFT	Fast Fourier Transform
FHA	Failure Hazard Assessment
GPS	Global Positioning System
GPU	Graphical Processing Unit
HDFS	Hadoop Distributed File System
HILS	Hardware In the Loop Simulation
HSI	Horizontal Situation Indicator
I/O	Input and Output
IMA	Integrated Modular Avionics
INS	Inertial Navigation System
IOSS	I/O Specific Segment
KF	Kalman Filter
OMS	Open Mission Systems
OS	Operational System
OSA	Open Systems Architecture

OSAL	Operating System Abstraction Layer
OSS	Operating Specific Segment
PCS	Portable Component Segment
POSIX	Portable Operating System Interface
PSSS	Platform-Specific Services Segment
RAM	Random Access Memory
RPA	Remote Piloted Aircraft
RTCA	Radio Technical Commission for Aeronautics
RTOS	Real Time Operational System
SAE	Society of Automotive Engineers
SDR	Software Defined Radio
SLOC	Single Lines of Code
SSA	System Safety Assessment
UAV	Unmanned Air Vehicle
UCS	Future Airborne Capability Environment
UoC	Unity of Conformance
USAF	United States Air Force
USDoD	United States Department of Defense

SUMMARY

LIST OF FIGURES 7

LIST OF TABLES 8

LIST OF LISTINGS 9

1 INTRODUCTION 13

1.1 Main Objective 15

1.2 Secondary Objectives 15

1.3 Document Structure 15

2 BACKGROUND 16

2.1 Avionics Certification Considerations 16

2.2 Federated vs. Integrated Avionics Systems 20

2.3 Open System Architectures 23

2.4 FACE – Future Airborne Capability Environment(TM) 24

3 RELATED WORKS 30

4 TOOLS AND METHODS 35

5 RESULTS AND DISCUSSION 47

6 CONCLUSION 59

REFERENCES 61

1 INTRODUCTION

Development of real-time embedded systems for avionics usage has evolved throughout the years, since hardware federated systems – where each functionality was implemented by a single device or by a dedicated electronic card packed together in an avionics cabinet – until Integrated Modular Avionics (IMA) systems. In IMA, functionalities are implemented by real-time software applications running over a shared hardware platform using a Real-time Operating System (RTOS) (WATKINS, 2007), providing a higher degree of standardization.

IMA introduction through DO-297 standard was important to enhance aspects crucial to aviation such as size and weight, electrical connections, ease of maintainability and upgrade possibilities, among others. This was possible because IMA allowed replacing several different instruments – each one dedicated to a specific purpose – in cockpit or avionics bay, like DME, HSI, Attitude Indicator, Altimeter, etc., and combine them in more dense architectures that supported different applications criticality. For instance, using a computer & display combination or even with smart displays capable of acquiring, processing and displaying navigation data in the same device. These key benefits were soon understood by the industry and operators, and therefore lead to a quick adoption of IMA systems in civil aviation market.

Nevertheless, the expected degree of modularization upon implementation of IMA-based systems running over commercial off-the-shelf (COTS) RTOS was not completely successful. Although it allows different applications to be independently implemented by different teams or even different companies, it still requires an entity (usually a system integrator) to know all application interfaces (that are not standardized) in order to integrate them to run over a shared hardware with temporal and spatial separation, thus assuring the proper behavior of each application. Furthermore, these applications are still tied to that specific hardware, preventing its re-use to be a straightforward activity. Finally, airworthiness certification is granted to IMA systems as a whole set and not to a specific software application. Thus, leading to increased costs, time and complexity when reusing or improving a software application that was part of an IMA certified system in another system or platform (LEWIS, 2003).

In order to overcome the remaining drawbacks that affected the development life cycle of both civil and defense embedded avionics systems, a new step was necessary. The first

movements were taken by defense contractors and end-users when they started pursue the adoption of Open Systems Architectures (OSA) towards an approach that could improve reuse of hardware and software components and services, reduce time-to-deployment, integration/acquisition costs and integration risks, therefore allowing affordable systems capability evolution, avoiding vendor lockdown and promoting innovation (TOKAR, 2017). US contractors are already exploring this kind of architecture: FACE and Open Mission Systems (OMS) take into consideration the potential business models, in order to promote an ecosystem that will enhance competition innovation, therefore leading to more robust and effective solutions at more competitive costs that create systems with interoperable and reusable components.

However, adding a conformance or standardization layer to a given application, either during design of a new application or while porting or adapting a legacy application, may bring collateral impacts that could affect key performance aspects that are crucial for the application operational usage, like execution time, memory usage or latency. For certain system types, performance and modularity can be improved by proper software architecture style selections (GALSTER, 2010). Nevertheless, for most embedded applications it is not possible to rely on constant hardware upgrades in order to improve their performance. For instance, avionics and unmanned air vehicle (UAV) applications have strict space and power consumption limitations, and a negative impact on performance may jeopardize system feasibility.

This dichotomy between enhancing portability/standardization at the price of degrading performance is not new in computing systems and affects several application types, from virtual machines used in cloud computing (SHAFER, 2010) to data types for scientific computing (DRAGAN, 2005).

In order to further explore this subject focusing in avionics domain, this work evaluates the performance of key characteristics on embedded systems while using two versions of a library: implemented with and without compliance to an Open Systems Architecture (OSA). In this case, it was used Future Airborne Capability Environment (FACE) (THE OPEN GROUP, 2016) as OSA case study, running over the same Operating System (OS). This evaluation is performed in two different hardware platforms in order to investigate the impact introduced by the OSA standard interface implementation running on different platforms.

1.1 Main Objective

The goal of this work is to quantify the impact (if existent) on performance while using FACE as an Open Systems Architecture for avionics mission applications. From the available works found in the literature, this is the first work containing experimental results to analyze such performance impact. Additionally, this data supports trade-off analysis that need to be considered while adopting this architecture in order to do not jeopardize the system feasibility.

1.2 Secondary Objectives

- Review literature on Open Systems Architectures adoption on real-time embedded systems for Avionics Defense Applications;
- Review literature on applicable performance metrics for real-time embedded systems;
- Retrieve performance metrics for a given example real-time application;
- Retrieve performance metrics for the example real-time application adapted to OSA framework
- Analyze and discuss performance metrics regarding performance optimization

1.3 Document Structure

This work is organized as follows: Section 2 brings a background about Avionics Systems & Certification, Open Systems Architectures and FACE as case study. Section 3 overviews related works. Section 4 details the methods and tools to perform the proposed study. Section 5 presents and discusses the experimental results and, finally, Section 6 concludes this work, providing directions for future works.

2 BACKGROUND

Avionics industry is very conservative regarding new technologies and processes, adopting them only when their benefits are clear and technology is mature enough to do not pose a threat to airworthiness safety. In this way, civil avionics market is more conservative than military avionics, where there is more room for testing and innovation. Therefore, new developments are usually introduced first in military platforms and later on, if successful and beneficial, civil market eventually adopts and may even enhance it. The next chapters bring an overview in some relevant aspects of avionics systems development and certification, regarding both military and civil platforms.

2.1 Avionics Certification Considerations

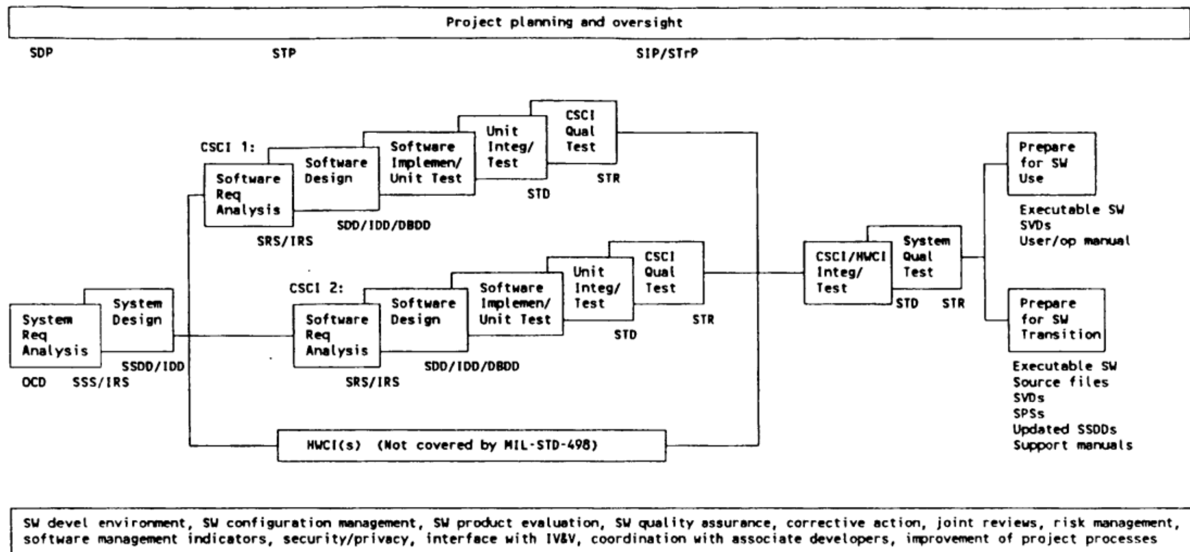
Most safety-critical embedded software development needs to comply with guidelines and standards in order to produce artifacts with a minimal level of quality, safety and maturity (LÖFWENMARK, 2014) to assure that embedded software will perform reliably in an airborne environment.

One of the first software development standards that was largely adopted by industry was DOD-2167. This standard was proposed by US Department of Defense in 1985 and aimed to provide a process and evidences to be delivered by companies willing to supply military avionics systems to US DoD contractors. As it became popular among industry, other western contractors started to adopt it in the same way or with minor tailoring. An updated version DOD-2167A was released in 1988.

Later in 1994, MIL-STD-498 (USA, 1994) was released superseding DOD-2167A and unifying it with DOD-2168 that dealt with software quality assurance issues. MIL-STD-498 was widely used not only by US but also for the majority of western defense contractors, as it enclosed a very comprehensive process for new and upgrade programs, as exemplified by the diagram on Figure 2.1. Eventually, MIL-STD-498 was the main basis for avionics software development guideline adopted by civil industry until the present days: DO-178B – Software Considerations in Airborne Systems and Equipment Certification (RTCA, 1992), developed by the safety-critical working group RTCA SC-167 of RTCA and WG-12 of EUROCAE. This standard is a guideline dealing with the safety of safety-critical software used in certain

airborne systems. The European agencies refer to this document as ED-12B, as registered by EUROCAE.

Figure 2.1 – One Possible Way of Applying MIL-STD-498 to Program Strategy



Source: MIL-STD-498 (1994)

During its system specification phase, every avionics software needs to be categorized according with a Design Assurance Level (DAL). The DAL is determined based on a safety assessment process and hazard analysis. A hazard is defined (FEDERAL AVIATION ADMINISTRATION, 2017) as a condition that could foreseeably cause or contribute to an accident, an unplanned event or series of events that results in death, injury, or damages to, or loss of, equipment or property.

A failure hazard assessment (FHA) is performed in order to verify the consequences of a failure condition in the system. The failures conditions categorization considers basically their effects on the aircraft, crew, and passengers as depicted in Table 2.1 defined in DO-178B (RTCA, 1992).

Table 2.1 – Failure Types and Consequences

Failure Type	*Failure Consequences*
Catastrophic	Failure may cause a crash. Error or loss of critical function required to safely fly and land aircraft.
Hazardous	Failure has a large negative impact on safety or performance, or reduces the ability of the crew to operate the aircraft due to physical distress or a higher workload, or causes serious or fatal injuries among the passengers. (Safety-significant)
Major	Failure is significant, but has a lesser impact than a Hazardous failure (for example, leads to passenger discomfort rather than injuries) or significantly increases crew workload (safety related)
Minor	Failure is noticeable, but has a lesser impact than a Major failure (for example, causing passenger inconvenience or a routine flight plan change)
No Effect	Failure has no impact on safety, aircraft operation, or crew workload.

Source: Author (2019).

Based on the definitions of Table 2.1, safety analysis tasks are accomplished in order to determine the software DAL and are required to be documented in system safety assessments (SSA). The relationship between function failure effects, DAL's and failure rate are represented in Table 2.2.

Table 2.2 – Design Assurance Levels per Failure Types.

Failure Condition	*Design Assurance Level*	*Maximum Failure Rate per Flight Hour*
Catastrophic	A	1.0E-9
Hazardous	B	1.0E-7
Major	C	1.0E-5
Minor	D	1.0E-3
No Effect	E	--

Source: Author (2019).

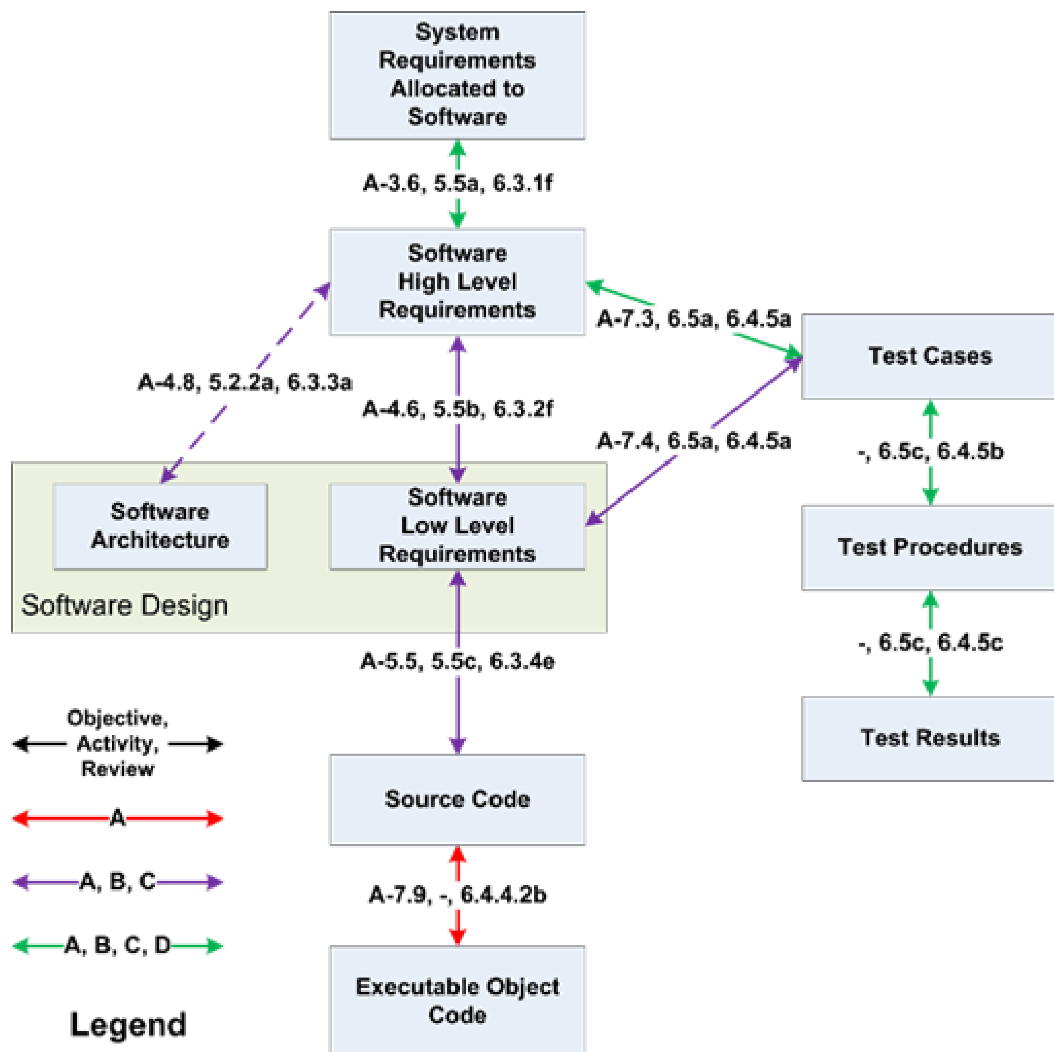
Furthermore, after specific safety analysis the hazard could be mitigated by system architecture aiming to decrease the DAL level, since development and certification costs greatly increase as the software criticality level is higher (POP, 2013).

DO-178B (RTCA, 1992) allows flexibility regarding different styles of software life cycles. This flexibility raises several abstraction aspects that could increase the complexity,

effort and cost depending how they are dealt. Independent of the aspects, all processes must have defined and documented the exit/entry criteria between development phases.

In order to comply with DO-178B, an avionic software application must follow the development phases described in the standard. Depending on the defined software DAL, some phases may be optional. Each phase generates a certification artifact. Figure 2.2 illustrates the trace between certification artifacts required by DO-178B/ED-12B.

Figure 2.2 – Relationship between DO-178B Certification Artifacts



Source: DO-178B (1992).

All these artifacts are evaluated by an independent certification organization, usually linked to a government department, which assesses if the development process was conducted in accordance with the standard and designed safety levels and then is able to approve its usage in airborne systems.

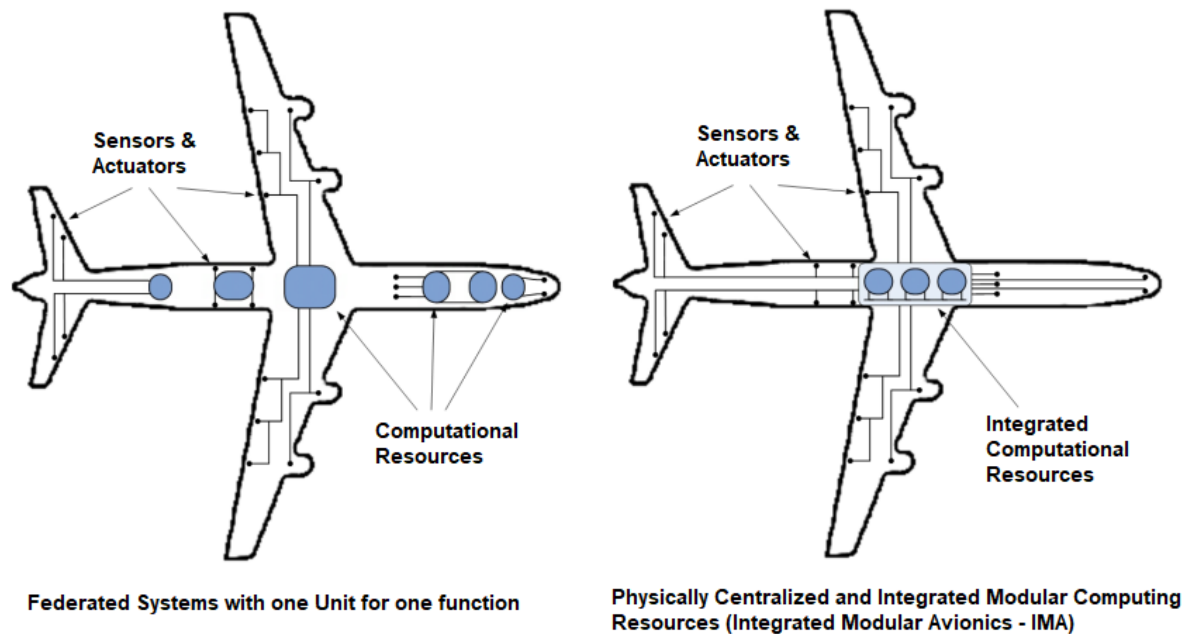
2.2 Federated vs. Integrated Avionics Systems

In the early days of electronic systems for aviation (avionics), each instrument or device had its own hardware developed as a stand-alone unit. As embedded electronics start to be more powerful and capable of processing more complex software programs, these avionic devices started to be able to deal with more attributions, but the whole avionics systems were still not working in an integrated way. This architecture was named federated systems, where each unit had its own responsibility and worked in an individual basis. An example of federated systems is shown in the left side of Figure 2.3.

Since the early 2000's, with the increase of processing capacity on embedded processors, civil avionics suppliers industry started to develop systems designed as software modules performing different tasks but embedded in a same hardware. This was an evolution of previous concept of having a same base hardware with several hardware cards, each one performing a dedicated task. This concept, defined as Integrated Modular Avionics (IMA), was standardized by DO-297 (RTCA, 2005) and brought a considerable improvement on avionics system design for civil platforms as depicted in the right side of Figure 2.3. Key benefits from IMA adoption are cost-effective upgrade paths, fast and efficient maintenance and avoiding premature obsolescence, to name a few (RTCA, 2005).

IMA concept allows a greater flexibility from a hardware perspective since it allows software applications processing data with different criticality levels over the same hardware platform. Moreover, these software applications (usually named Partitions), can communicate not only with other hardware devices connected to the IMA hardware through a data bus (e.g.: ARINC 429, RS-422, CAN) but also allows communication between partitions installed in the same hardware. This communication is performed through interfaces defined by an IMA system integrator.

Figure 2.3 – Federated vs. IMA Architectures



Source: Wolfig (2008).

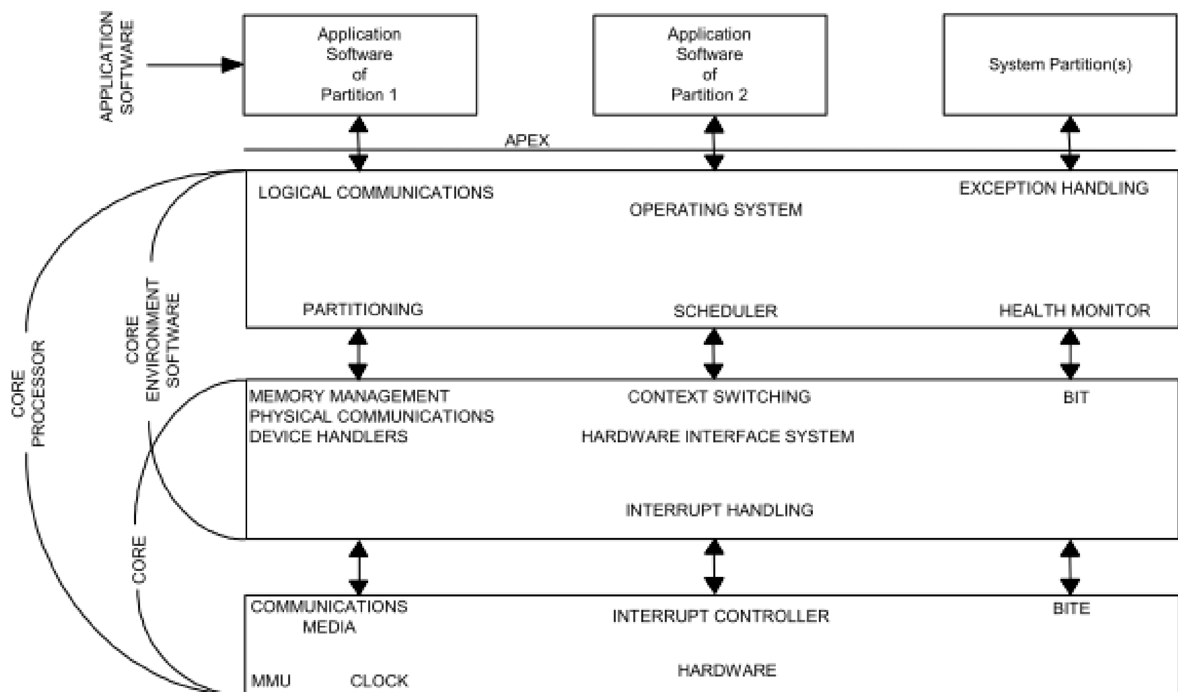
IMA systems are usually deployed over Commercial Off-The-Shelf (COTS) hardware, leaving most of dedicated and customized tasks to be performed by software and programmable hardware applications. The foundations of IMA design rely on the determinism of the combination of each application and the hardware over which it is running. Depending on the criticality of the function being implemented by an IMA embedded software application, it is mandatory to know the expected behavior of this implementation in such a way it will not interfere with other applications running over the same processor and sharing the same hardware resources (LÖFWENMARK, 2014).

The best way to assure this independence and non-interference between applications running over the same hardware is assuring the temporal and spatial separation between applications (WEILONG, 2014). Spatial separation means to assure that each application has its own memory area and this area will not be used by any other application except the one intended to use. Temporal isolation means that a given application will seize the hardware resources to execute only during a given pre-established amount of time and this amount will not be exceeded in order to not jeopardize the execution of other applications that will run using the same hardware resources.

Spatial separation is a goal that does not present major challenges to be achieved, since real-time operating systems (RTOS) compliant with ARINC 653 standard (ARINC,

2006) offer a robust tool set to assure the isolation of memory areas between applications (PRISAZNUK, 2008). However, temporal separation is a much more delicate issue to handle. This happens because in hard real-time applications, each task has hard deadlines to meet and the real-time operating system shall manage the scheduler in order to avoid any unexpected and not deterministic behavior. This is the key factor that allows applications of different severity levels (Design Assurance Levels as mentioned in previous chapter) to be executed in the same framework and therefore over the same hardware. An example of IMA software architecture using ARINC 653 is shown in Figure 2.4.

Figure 2.4 – ARINC 653 Software Architecture Example



Source: Prisaznuk (2008).

Certification costs are a major drive in an avionics development process, which may reach more than half of the full development costs. Therefore, it is a primary goal for the whole industry to pursue more effective approaches to obtain certification. One of the most important is to improve the reuse of previously certified items. IMA systems started to address this issue and indeed brought a considerable improvement when compared to hardware-federated systems. However, there is still issues that needs to be improved from IMA systems, like modularity and de-coupling from hardware. These issues were further explored in defense systems by Open Systems Architectures, detailed in next chapter.

An interesting parallel to notice is the opposite way as DO-178B evolved as a civil guideline based on military standard MIL-STD-498, Open Systems Architecture concept started to gain traction among defense contractors and industry based on the positive experience on DO-297 and Integrated Modular Avionics adoption by civil industry, authorities and operators.

2.3 Open System Architectures

The United States Department of Defense (USDoD) has been focusing efforts to find tools and technologies to accelerate the development and evolution of military application systems. Through internal directives (USA, 2015) (USA, 2017a) (USA, 2017b), it decided to embrace Open Systems Architecture frameworks.

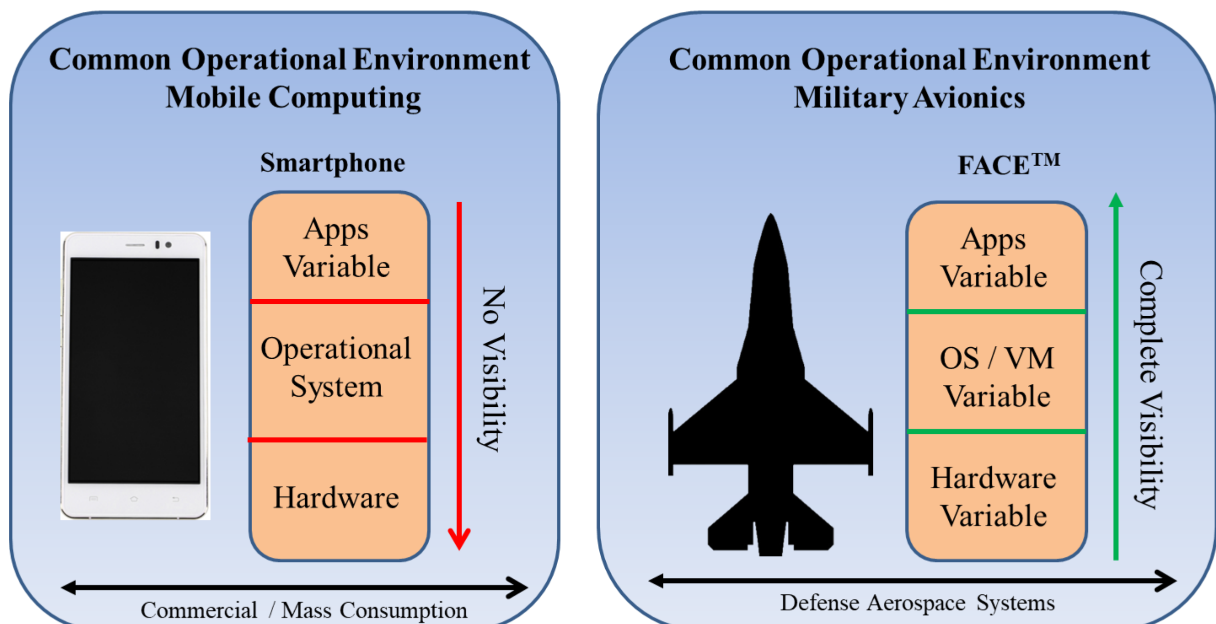
The concept behind standardized Open Systems Architecture (OSA) is to allow software applications development with well-defined and known common interfaces, standardized in such a way that allows an individual validation and certification process, thus assuring the software application being independent from hardware and operational system. OSA are based on reusable hardware and software components and services, in order to deliver enhanced and integrated capabilities at a lower life cycle cost, while avoiding vendor lock. Therefore, it enables affordable capability evolution and promotes innovation. It also allows an easier reuse of software applications between different end platforms and hardware.

According to (TOKAR, 2017), an open systems architecture is a system that employs modular design, uses widely supported and consensus-based standards for its key interfaces and is subjected to successful validation and verification tests to ensure the openness of its key interfaces. Focusing on military avionics systems, several OSA standards have been developed and fostered by USDoD, like Unmanned Systems Control Segment (UCS) Architecture (that later evolved into a library of SAE publications, headed by AS6512 (SAE INTERNATIONAL, 2016), Open Missions System Initiative (sponsored by USAF and based on Service Oriented Architecture and middleware) and the Future Airborne Capability Environment (FACE) (THE OPEN GROUP, 2017). This work focuses on FACE, which is further detailed below.

2.4 FACE – Future Airborne Capability Environment(TM)

The FACE Enterprise (THE OPEN GROUP, 2016) is comprised of stakeholders representing the government (Contractors), Industry, and Academia, all working towards a common goal of proposing, developing, and implementing software application developed, validated and certified in adherence to FACE Technical Standard (FACE-TS). This standard (THE OPEN GROUP, 2017) describes a standardized software Common Operational Environment (COE) that is hardware-agnostic, based on the successful experience of other mass consumption electronics items like smartphones, where different companies and entities develop software applications respecting a well-defined interface that runs over different COTS hardware types (TOKAR, 2017), depicted in left side of Figure 2.5. FACE-TS describes requirements for architectural segments with their associated modular software components and defines key interfaces that link the segments together. It also establishes a framework to enable the affordable acquisition of software systems that promotes rapid integration of portable capabilities across defense programs portfolio. FACE-TS specifies how to develop an open, modular, software environment for security, safety, or general purpose software COEs, shown in the right side of Figure 2.5.

Figure 2.5– Common Operational Environment Examples



Source: Author (2019).

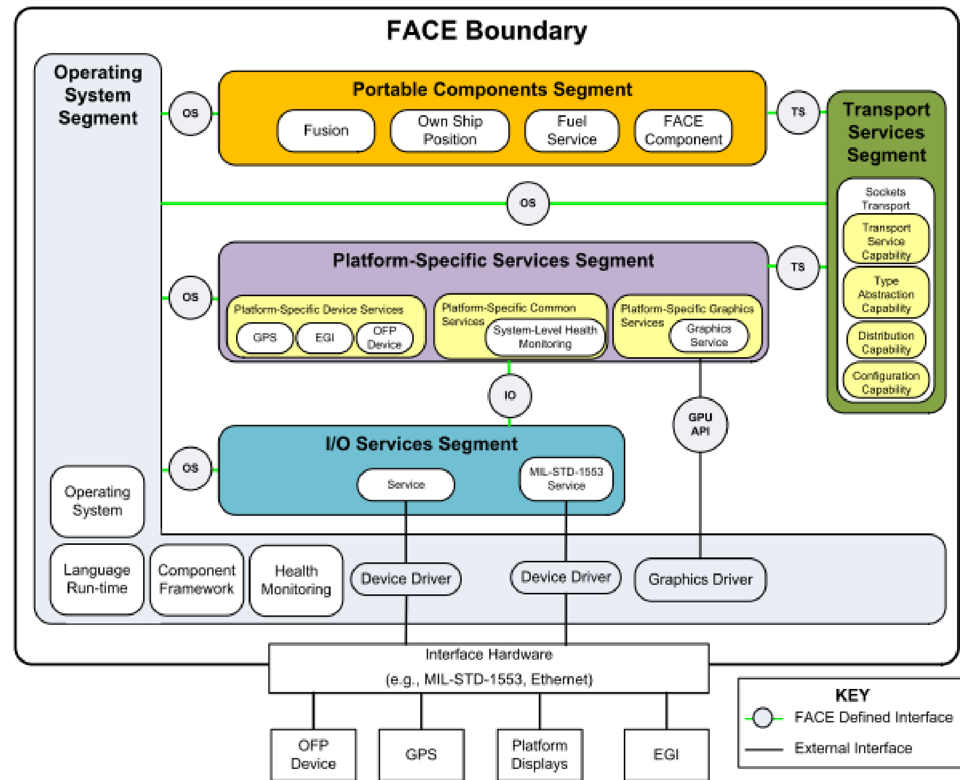
FACE business model also encourages innovation and competition. Any company willing to supply a FACE-conformant application can initiate a verification process aiming to

demonstrate adherence to FACE Technical Standard through usage of a Conformance Verification Matrix and to submit it to a Conformance Test Suite by a recognized FACE Verification Authority recognized by the FACE Certification Authority (CA). Once it completes these steps, an application can finally be registered as FACE Certified UoC (Unit of Conformance) in FACE Registry and then to be found as a reusable product in a FACE conformant system.

The architecture modularity and adherence to the standard interfaces as proposed by FACE allows several different teams or companies to develop designs for a certain functionality (layers), being able to compete or team together in order to deliver a complete functionality, such as an Embedded GPS/INS system (WALLACE, 2017). Another remarkable feature on FACE is to rely on *de facto* standards widely adopted and established by industry (THE OPEN GROUP, 2017) instead of proposing a completely new set of operating systems, data buses and communication protocols. This facilitates the adoption of FACE Architecture, as exemplified by Figure 2.6 on new developments, since companies and teams are already familiar with such standards, such as MIL-STD-1553, ARINC 429, CAN, Ethernet, APEX, ARINC 653, etc.

The FACE approach allows software-based capabilities to be developed as components that are exposed to other software components through defined interfaces. It also provides for the reuse of software across different hardware computing environments, thus allowing the same application to be used in several platforms with minimal or practically no changes.

Figure 2.6 – FACE Architectural Segments Example



Source: The Open Group (2017)

The FACE-TS requirements for architectural segments with their associated modular software components, and defines key interfaces that link the segments together. The FACE Reference Architecture is defined by five segments and three key interfaces.

The FACE-TS segments are resumed below as described in FACE Overview document (THE OPEN GROUP, 2016).

1. **Operating System Segment (OSS):** OSS is where foundational system services used by all of the other segments and vendor-supplied code reside. The OSS provides and controls access to the computing platform for the other FACE segments. The OSS manages the execution of software associated with the other FACE segments and hosts various operating system and low-level health monitoring interfaces. The OSS can also optionally host external networking capabilities, programming language run-times, and component framework interfaces.
2. **Input/Output Services Segment (IOSS):** IOSS is where normalization of interface hardware device drivers resides. This normalization is achieved using a

set of adapter design patterns that individually communicate to a vendor-supplied driver and then convert that data to a standardized FACE interface. The I/O Services within this segment provide a bridge for subsystem data between the interface hardware to the Platform-Specific Services Segment (PSSS).

3. **Platform-Specific Services Segment (PSSS)**: PSSS is comprised of sub-segments including Platform-Specific Device Services, Platform-Specific Common Services, and Platform-Specific Graphics Services. Device Services is where translation between platform-unique Interface Control Documents and the FACE Data Model occurs. Common Services is comprised of higher-level services including Logging Services, Centralized Configuration Services, Device Protocol Mediation Services (DPMS), Streaming Media, and system-level HMFM. Graphics Services is where presentation management occurs. Examples might include software that translates radar proprietary data format or Electronic Support Measures to data that is understandable to all FACE components above the PSSS. If the software being developed interacts directly with a physical device, it most likely belongs in the PSSS. If the device does not implement a standards-based interface, the software definitely belongs in the PSSS.
4. **Transport Services Segment (TSS)**: TSS is where communication services reside. Data Distribution occurs between and within software residing in either the Portable Component Segment (PCS) or the PSSS. Other communication services include Quality of Service, Data Transformation, Paradigm Translations, and Message Association. Examples might be Data Distribution Service (DDS) or CORBA implementations and converting Kelvin to Fahrenheit.
5. **Portable Components Segment (PCS)**: PCS is where software providing mission-level capabilities or business logic resides. Capabilities contained within the PCS should remain agnostic from hardware and sensors (as well as logical topologies of subsystems; e.g., dual or triplicated buses), free of any data transport mechanism or operating system specifics. This maximizes a capability’s portability and interoperability across varying platforms. Examples of a PCS component might be data fusion and calculate own ship position.

Finally, the three FACE-TS key interfaces as described in FACE Overview document (THE OPEN GROUP, 2016) are summarized below.

1. **Operating System Segment Interface** – This interface provides a standardized means for software to utilize the services within the operating system and other capabilities related to the OSS. This interface is provided by software in the OSS to software in other segments. This interface includes ARINC 653, POSIX, and HMFM APIs. This interface is optionally allowed to include Internet networking capabilities, programming language run-times, and component frameworks.
2. **I/O Services Interface** – This interface provides a standardized means for software to communicate with interface hardware device drivers. This interface is provided by software in the IOSS to software in the PSSS. It utilizes messages formatted using the I/O Message Model.
3. **Transport Services Interface** – The TS Interface provides a standardized means for software to utilize communication services provided by the TSS. This interface is provided by software in the TSS to software in the PSSS and PCS. It utilizes messages formatted using the FACE Data Architecture.

In order to address different levels of application criticality, FACE Technical Standard defines three Operating System Profiles (Security, Safety (based and extended), and General Purpose) as subsets of POSIX 1003.1b (IEEE, 2017) to run as FACE Operating System Segment (OSS), where a stricter profile are always a subset of a broader profile. Their characteristics are detailed below:

- **Security Profile**: aimed to applications where security is a key feature, it requires full ARINC 653 compliance and allows a more restrict subset of 136 POSIX methods, constraining the OS APIs to a minimal useful set. This allows the assessment of high-assurance security functions executing as a single process.
- **Safety Profile**: with a less restrictive approach, it still requires full ARINC 653 adherence, although methods to delete or close OS objects are not available. This profile is divided in two sub-profiles:
  - Basic Safety Profile, in which 246 POSIX methods are included and only single process is allowed;
  - Safety Extended Profile, in which 335 POSIX methods are included and multiprocess & multithread are allowed.

- **General-Purpose Profile**: the least constrained profile, it supports 812 POSIX methods and allows applications that do not have safety, real-time or determinism requirements to have a richer POSIX support.

It is important to highlight that although FACE OS Profiles support POSIX calls, it does not implement the complete set of POSIX methods (over 1300 methods), since based on the industry experience on avionics certification it is considered that some of the POSIX methods like *strtok()* were unsafe for multi-threading programming (BLOOM, 2018).

Table 2.3 – Available POSIX methods per FACE profile.

FACE Profile	*Amount of POSIX methods*
Security Profile	136
Basic Safety	246
Safety Extended	335
General-Purpose	812

Source: Author (2019).

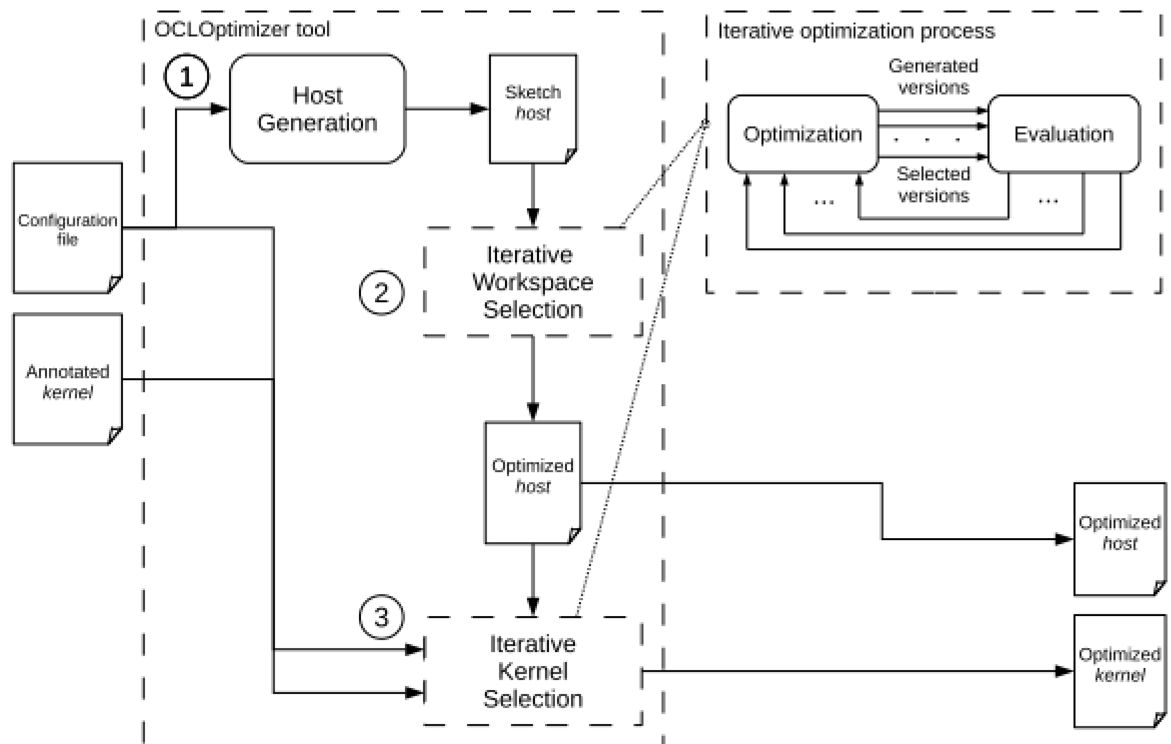
The FACE Enterprise is being successful as an Open Systems Architecture implementation (THE OPEN GROUP, 2017), although nowadays it is restricted to United States defense market. Nevertheless, US defense market is the largest global player (SIPRI, 2019) and given the past history of US standards and initiatives influencing other markets and becoming *de facto* standards (e.g.: MIL-STD-498 for software development and documentation). Therefore, it worth to understand and eventually engage in development efforts considering FACE architecture to leverage from its benefits, lessons learned and potential future synergy and integration with defense systems developed by US-based defense companies. Moreover, the arrangement between FACE stakeholders in defense industry (suppliers), government (contractors) and academia (innovation and tools) have an important similarity with the Triple Helix concept researched and defined by Brazil (ETZKOWITZ, 2017) and fostered by Brazilian Ministry of Defense and Brazilian Army (EXÉRCITO BRASILEIRO, 2019), which may open possibilities for Brazilian players enter into US defense market in the future. For the reasons listed above, this work selected FACE as open system architecture example to be evaluated.

3 RELATED WORKS

The search for portability and open common interfaces is an objective pursued for a long time in Computing Systems in order to facilitate integration in different platforms and thus reduce development costs and shorten the life-cycle deployment. A good example of solution that present satisfactory results in dedicated implementations, but falls short on performance when a common interface was implemented over it are Accelerator Processors or Graphical Processing Units (RUL, 2010). Each type of accelerator processor uses a particular interface to achieve the maximum performance, causing its interfaces to be specific for each vendor or architecture. This forces software developers to adapt their applications each time an accelerator is changed. In order to bridge this gap, OpenCL language was defined to bring a common interface for accelerators and therefore facilitate software development. However, although this initiative brought enhancements for functional portability, it failed on bringing performance portability, since the optimization tailored to each architecture on previous development was not achieved when using the common interface (KOMATSU, 2010).

One of the drawbacks on implementing a common interface in order to support a variety of hardware platforms is possibly the impact on performance (e.g.: worst case execution time, memory consumption, latency, etc.), namely the overhead to implement wrappers and adapting internal interfaces to standard interfaces. A similar phenomenon affected OpenCL language, aimed to standardize the implementation of software over different GPU and CPU platforms. As described in (KOMATSU, 2010), the generic implementation did not reach the same performance levels of dedicated applications tailored to a specific hardware platform. Further works focused on OpenCL compiler to optimize code generation for each specific GPUs. Even though, it was still necessary that manual tailoring and optimizations on the compiler options were performed in a trial-and-error basis for each hardware in order to achieve performances comparable to HW-specific applications (KOMATSU, 2010). It means, although technically feasible alternatives exist to overcome this performance gap (e.g.: auto-tuning), these alternatives may be extremely time-consuming, therefore jeopardizing the benefits obtained from code portability. Due to this fact, research efforts have been made focusing to perform this tuning in a more automated way like OCLOptimizer tool depicted in Figure 3.1 (FABEIRO, 2015), although it still requires an iterative process and optimization directives that need to be informed through configuration files in a case-by-case basis.

Figure 3.1 – General workflow of OCLoptimizer



Source: Fabeiro (2015).

A similar empirical study was conducted by (VARBANESCU, 2015). This study considered three different graph algorithms (with eight input datasets each) implemented in OpenCL, running over three different hardware platforms (CPU & GPU) and compared their execution for each implementation. Eventually the study showed that portability can provide performance improvement by allowing execution of the same solution in different combinations of CPU and GPU processors. Still, a performance verification on a trial-and-error basis was required to find the best fit among CPUs and GPUs to maximize performance.

However, the performance improvement presented in (VARBANESCU, 2015) apparently contradicts the results in (KOMATSU, 2010), since it showed a considerable performance decrease while comparing hardware dedicated applications with generic ones. The two studies differ in the use of more than one processor for each solution in (VARBANESCU, 2015), which allows the use of the strengths of each processor in different computations and has the ability to increase the general performance.

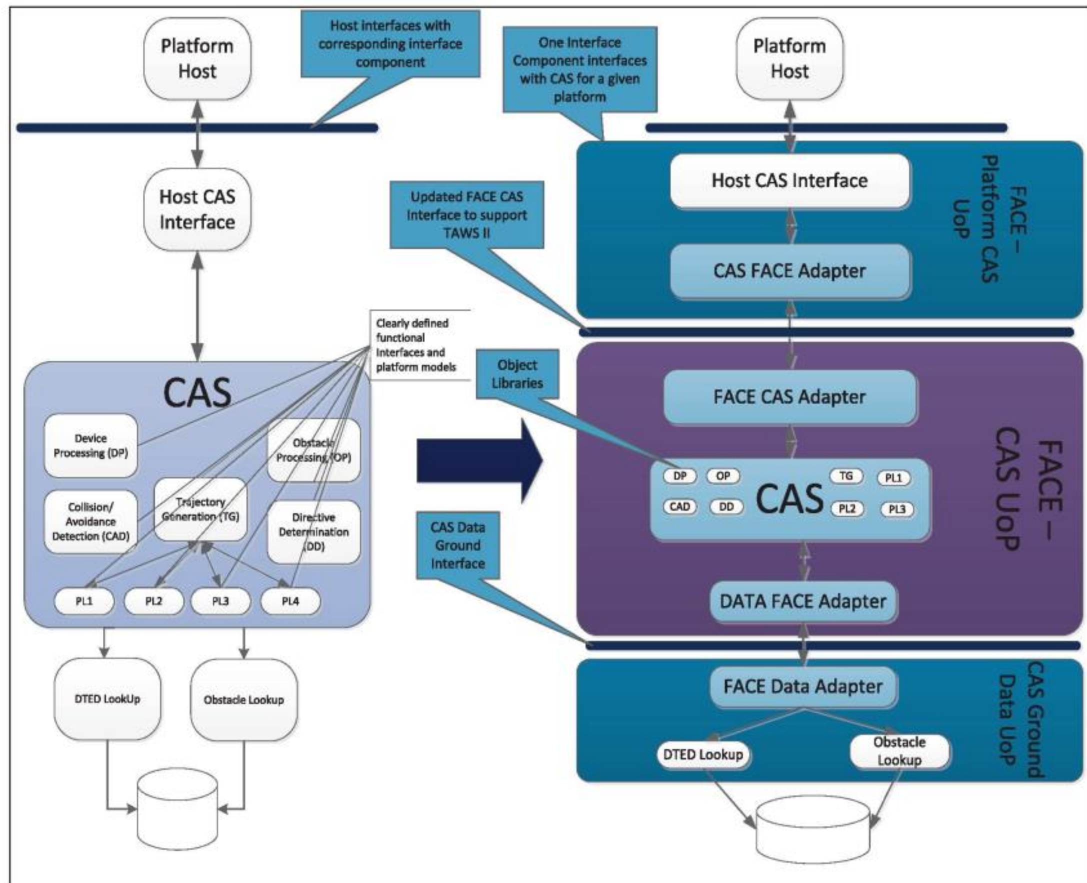
Another example of performance drawback to achieve portability is also faced on HDFS distributed file system for Hadoop (a popular open-source implementation for analysis

of large datasets). As HDFS is written in Java to achieve portability across heterogeneous hardware and software platforms, (SHAFER, 2010) uncovered several performance issues due to factors listed below.

1) Architectural bottlenecks: scheduling delays are posed by Hadoop implementation because disk access pattern is periodic instead of streaming and file reads are serialized instead of decoupled and pipelined.
2) Portability limitations: some performance-enhancing features of native filesystems are not explored since Java does not offer these features in a platform-independent manner, therefore optimizations available in native filesystem may remain unused, although available.
3) Implicit portability assumptions that eventually proved not to be true: although functionally Hadoop provides portability, performance is highly dependent on the native platform implementation, specially OS I/O scheduler and native filesystem.

Regarding FACE implementation, a good overview is provided by a legacy Collision Avoidance System software application developed by and for US Navy that was adapted for FACE framework and deployed further on to six different platform is detailed by (BRABSON, 2015). This evolution, depicted by Figure 3.2, allowed the development team to decrease development cycles and integration issues. This work mentioned the addition of an execution overhead when the new architecture was adopted, however such overhead was not quantified.

Figure 3.2 – CAS product architecture for FACE



Source: Brabson (2015)

Additionally, (HAN, 2014) performed an extensive comparison analysis through quantitative tests that aimed to verify performance difference on several partitioning design for ARINC 653 systems. The tests used a real avionic application exercising three partitioning approaches: at kernel level, at user level and through virtual machine using HILS (*Hardware In-the-Loop Simulation*). Although it used only one hardware platform, it uses a similar environment architecture: application, item under measurement (operating system in this case) and hardware.

Table 3. below shows a comparison summarizing the related works mentioned above and the scope of this work.

Table 3.1 – Scope comparison between related works and this work.

Author	*Portability*	*Avionics*	*Embedded Systems*	*Performance Evaluation*
Rul	X		X	
Komatsu	X		X	X
Varbanescu	X		X	X
Shafer	X			X
Brabson		X	X	
Fabeiro	X		X	X
Han		X	X	X
(this work)	X	X	X	X

Source: Author (2019).

As reviewed in the related works mentioned above, functional portability is a known issue and well handled through common interfaces and standardization. However, performance portability is still an open issue that requires dedicated advances techniques to be applied in order to handle it. Although for OpenCL and GPU processing there is active research ongoing and producing results, for avionics open systems architectures this problem is not quantitatively identified already and therefore not handled yet. This is the gap to be addressed by this work.

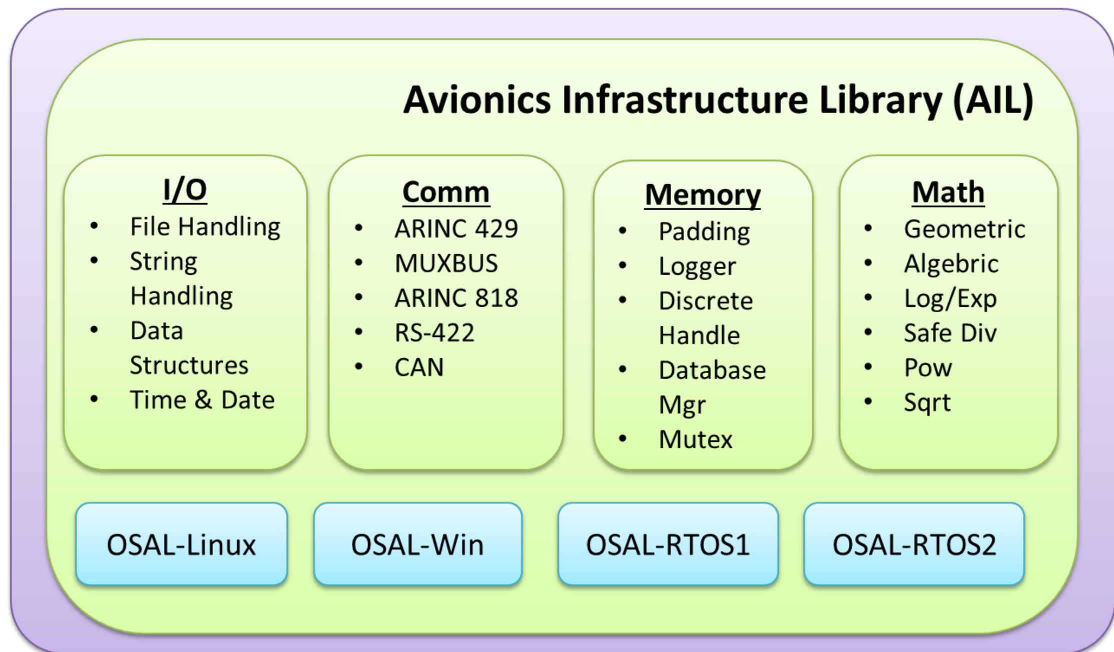
4 TOOLS AND METHODS

The initial approach considered to quantify the possible overhead introduced by FACE implementation was to perform it in a similar way as the work described in (BRABSON, 2015). It means to have a stand-alone, end-to-end defense application currently implemented in a non-modular way to use as a baseline application and then redesign and adapt it to be conformant to Open Systems Architecture (i.e.: FACE in this case), while keeping all its original features.

However, it was not possible to select an end-to-end defense application that could be openly disclosed without jeopardizing confidentiality, intellectual property or security constraints. Eventually, the solution was to do in an opposite way as performed in (BRABSON, 2015). It means to use as baseline a library that provides general infrastructure services to more complex applications through simpler, well-known software functions that is already implemented adherent to FACE and then re-implement it as a simplified version without the modularity and abstraction layers required by FACE standard.

The selected library is named Avionics Infrastructures Library (AIL) (Figure 4.1) and is used in several end-to-end avionics defense applications. It implements a variety of basic level services of mathematical, data manipulation functions and algorithms widely used in avionics embedded systems, such as memory (and file) handling, binary operations (byte-wise and word-wise), communication protocols handling (e.g.: ARINC 429, MUXBUS, etc.) such as raw to engineering conversion, validity verification and padding/unpadding.

Figure 4.1 – AIL structure



Source: The Author (2019)

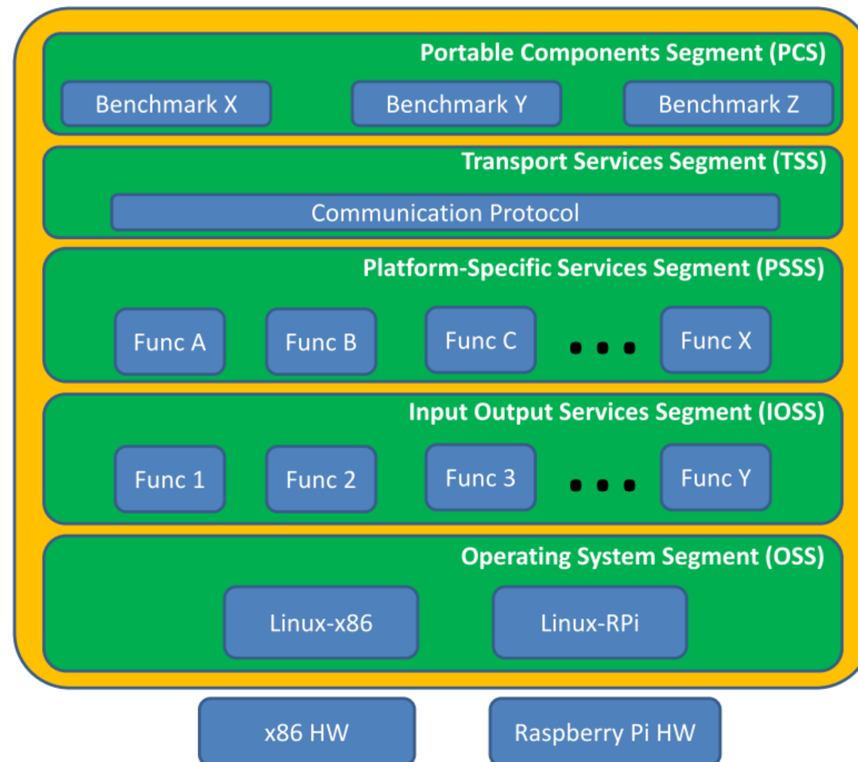
This library is originally designed and implemented as FACE Technical Standard 3.0 (THE OPEN GROUP, 2017) and is structured as a set of stand-alone functions not directly linked to a top-level application (or PCS – Portable Component Segment as per FACE architecture). The library is implemented using FACE General Purpose Profile and is composed by approximately 100 services and is implemented in ANSI C and has a total of more than 3500 single lines of code (SLOC). Since it is a proprietary library, its source code can not be included or displayed in this text.

AIL is organized considering to match three layers per FACE architecture:

1) A portion relative to OS Abstraction Layer categorized as Operating Specific Segment (OSS) in FACE architecture;
2) A portion dealing with functions provided by AIL categorized as Platform-Specific Services Segment (PSSS);
3) An I/O Specific Segment (IOSS) portion handling with input and output mechanisms (namely file handling).

The complete architecture division is depicted in Figure 4.2.

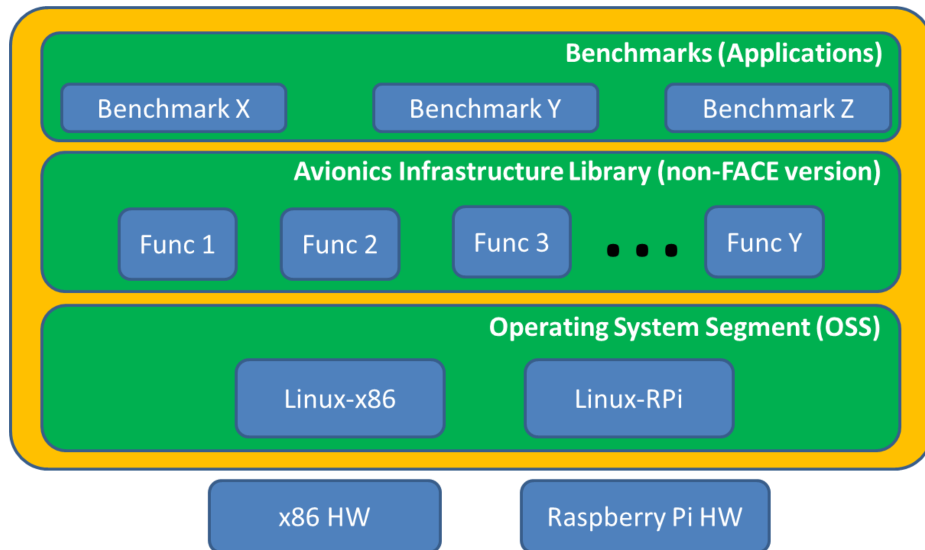
Figure 4.2 – AIL FACE Architecture



Source: The Author (2019)

In order to perform the comparison to the FACE adherent AIL, a second version of the library was implemented specially for this work using the original AIL as a baseline, however converted to a flat architecture (Figure 4.3), it means, without the abstraction layers required by FACE Technical Standard 3.0 (THE OPEN GROUP, 2017). This non-FACE implementation was also performed in ANSI C, where all required POSIX functions were directly implemented as standard system calls, without modularization, although still providing the very same functionality. This simplified non-FACE implementation was used as reference to obtain performance values to compare against FACE implementation performance and therefore assess the impact open systems architecture implementation over the two versions of the same library providing the same functionality. It resulted in a library of approximately 40 services implemented by more than 1000 single lines of code (SLOC).

Figure 4.3 – AIL non-FACE (flat) architecture



Source: The Author (2019)

An illustrative example using code snippets on FACE implementation is showed in Listing 4.1 until Listing 4.5. There, it can be seen how each FACE segment handles data in a different level of abstraction, keeping the modularity and allowing updates and changes in lower levels without affecting the business logic. In Listing 4.1, lines 5 to 8 show part of the business logic, while Listing 4.2 line 6 shows data receiving, regardless of the transmission channel, which is handled in PSSS level (Listing 4.3). Lines 3 and 7 on Listing 4.4 show the handling of data retrieving and finally Listing 4.5 show the actual activation of communication channels that is OS-dependent.

In Listing 4.6, a similar logic without modularity is showed, where the business logic is mixed with communication (see lines 4 to 6) and detailed design (lines 13 to 17) in the same layer/file, making future customizations or upgrades harder. These parts can be compared to the separate modules in Listing 4.1 to Listing 4.5, where the business logic is separate from the other modules, as described above.

Listing 4.1 – FACE PCS Segment Source Code

```
/*======================= PCS ===============================*/
01 void PCS_GetAircraftID(UINT32 uiInstance)
02 {
03  /*...*/
04  uiICAOId = TSS_GetAircraftIDMsg(uiInstance, ICAO_ID);
05  if(uiICAOId == AC_B73M)
06  {
07         /*...*/
08  }
09  /*...*/
10}
```

Source: The Author (2019)

Listing 4.2 – FACE TSS Segment Source Code

```
/*======================= TSS ===============================*/
01 UINT32 TSS_GetAircraftIDMsg(UINT32 uiInstance, UINT32 uiID)
02 {
03  /*...*/
04  if((uiInstance>g_uiMinInstance)&&(uiInstance<g_uiMaxInstance))
05  {
06         PSSS_ReadADS_B_In(&uiICAOId, ICAO_ID);
07  }
08  /*...*/
09 }
```

Source: The Author (2019)

Listing 4.3 – FACE PSSS Segment Source Code

```
/*====================== PSSS ===============================*/
01 UINT32 PSSS_ReadADS_B_In_UDP(UINT32 uiInstance, UINT32 uiID)
02 {
03  /*...*/
04  if (ADS_B_In == STATUS_ACTIVE)
05  {
06          return(IOSS_ReadADS_B_In_UDP(ICAO_ID));
07  }
08  else if (ADS_LocalServer == LOCAL)
09  {
10          return(OSS_ReadFile(g_iLocalADSFileID));
11  }
12  else
13  {
14          return(ERROR_ID);
15  }
16  /*...*/
17 }
```

Source: The Author (2019)

Listing 4.4 – FACE IOSS Segment Source Code

```
/*====================== IOSS ===============================*/
01 UINT32 IOSS_ReadADS_B_In_UDP(UINT32 uiInstance, UINT32 uiID)
02 {
03  if (ADS_B_In == STATUS_ACTIVE)
04  {
05          return(OSS_ReadUDPSocket(ICAO_ID));
06  }
07  else if (ADS_LocalServer == LOCAL)
08  {
09          return(OSS_ReadFile(g_iLocalADSFileID));
10  }
11  else
12  {
13          return(ERROR_ID);
14  }
15  /*...*/
16 }
```

Source: The Author (2019)

Listing 4.5 – FACE OSS Segment Source Code

```
/*====================== OSS ==============================*/
01 UINT32 OSS_ReadUDPSocket(UINT32 * buffer)
02 {
03  /*...*/
04  ssize_t count = recvfrom(fd,buffer,
                  sizeof(buffer),0,
                  (struct sockaddr*)&src_addr,
                  &src_addr_len);
05  if (count==-1)
06  {
07         OSS_HandleError("%s",strerror(errno));
08  }
09  else
10  {
11         handle_datagram(buffer,count);
12  }
13  /*...*/
14}
```

Source: The Author (2019)

Listing 4.6 – Non-FACE Equivalent Source Code

```
01 void * GetAircraftID(UINT32 uiSize)
02 {
03  /*...*/
04  if(uiADS_B_ServerType == COMM_UDP)
05  {
06         ssize_t count = recvfrom(     fd,
                                     buffer,
                                     sizeof(buffer),0,
                                     struct sockaddr*)&uiADS_B_src_addr,
                                     &uiADS_B_src_addr_len);
07    if (count == -1)
08    {
09      die("%s",strerror(errno));
10    }
11    else
12    {
13      if((buffer[ICAO_ID] & 0x7F) == AC_B73M)
14      {
15        /*...*/
16      }
17    }
18  }
19  else if(uiADS_B_ServerType == COMM_LOCAL)
20  {
21    /*...*/
22  }
23}
```

Source: The Author (2019)

Since AIL *per se* is not capable to play the role as an avionic application, in order to exercise its both versions (FACE and Non-FACE), four applications were chosen as benchmark top-level application as Portable Component Segment (PCS) per FACE architecture. These benchmark applications were modified only to replace system calls by AIL services whenever possible. Their end functionalities were completely preserved. The selected benchmarks are listed below:

- Fast Fourier Transform (FFT) (GUTHAUS, 2001): This application constantly calculates FFT on a 32768 data items array containing 8 waves of random amplitude and frequency. Fast Fourier Transform is used in digital signal processing

to find frequencies contained in a given input signal, being heavily used in digital communications, Software Defined Radio (SDR), radar target detection (YU, 2012), and other aerospace applications. This benchmark application uses mainly the memory manipulation and mathematical functions of AIL such as trigonometric functions.

- Kalman Filter (KF): KF (WELCH, 1995) is used in several aerospace applications such as navigation, position estimation, radar target tracking and data prediction. In this work, this application was configured to reduce a noise inserted in a fixed value represented by a Z axis acceleration measurement with a random error added (limited to 9% of the ideal value) during 30 iterations. This application uses more intensively the memory manipulation AIL services.
- Dijkstra Algorithm (DIJ) (GUTHAUS, 2001): This benchmark constructs a large graph in an adjacency matrix representation and then calculates the shortest path between every pair of nodes using repeated algorithm executions. In this example it was used a fixed large graph in an adjacency matrix representation (100x100). Dijkstra algorithm is a well-known solution to the shortest path between nodes in a graph problem and completes in $O(n^2)$ time. In the avionics domain it has several applications such as in navigation systems. This application uses more intensively the memory manipulation and file handling AIL services.
- CRC32 (GUTHAUS, 2001): This benchmark performs a 32-bit Cyclic Redundancy Check (CRC) on an input data stream. CRC checks are often used to detect errors in data transmission and file storage/reading consistency checks. In this work, the CRC calculation was performed on an example PCM sound file of 25.3MB. This application uses more intensively the memory manipulation and file handling AIL services.

These general-purpose algorithms and functions were selected as benchmark implementations since they do not have confidentiality constraints as other defense applications but still perform computation similar to the ones used in defense embedded applications. As it could be verified to the moment when this experiment was conducted, this is the first public work bringing quantitative analysis of the impact on FACE adoption on embedded systems.

The experiment was conducted using both AIL versions implemented over Linux OS and using two different hardware platforms in order to verify the FACE implementation impact dependency regarding the platform used under OSS segment.

Both platform types selected are popular in embedded system applications in their segments (graphical/general purpose processing for x86 and compact/low power processing for Raspberry Pi).

The first platform was an Intel i5-2430M quadcore processor running @ 2.4GHz with 64kB L1 cache, 512kB L2 cache and 3MB L3 Cache, 6GB RAM running Linux 4.15.0-50-generic. This x86 platform was selected as baseline given its wide usage range from consumer electronics to military mission system, such as tactical digital maps and ground-control station systems (MCHALE, 2019).

Figure 4.4 – x86 Platform Setup (illustrative)

Source: VR (2019)

The second platform used was a Raspberry Pi 3 Model B card with Quad Core 1.2GHz Broadcom BCM2837 64bit CPU and 1GB RAM running Linux Raspbian version 4.9.80-v7+. It was chosen as an example of small and simple, yet complete, hardware that is currently popular for usage in embedded applications such as Remote Piloted Aircraft (RPAs) (SANTOS, 2017) and aerial image processing.

Figure 4.5 – Raspberry Pi 3 Platform Setup (illustrative)

Source: RASPBERRY PI (2019)

For each AIL implementation, the application performance was measured using Linux native tool PERF (DIMAKOPOULOU, 2016). This tool allows efficient and non-intrusive performance monitoring and profiling on Linux, allowing access to processor performance counters and other hardware events that are monitored and stored by perf-event Linux subsystem. Five parameters available on Perf statistics were selected to assess the overall system performance while running the example applications:

- Task Clock
- Cycles
- Instructions
- L1 Cache References
- L1 Cache Misses

Task Clock, which eventually is directly linked to execution time, was selected as it is a key parameter for hard real-time systems since it is a prerequisite to perform scheduling analysis and thus assure system feasibility to meet required deadlines (WILHELM, 2008). Number of execution cycles and number of executed instructions were selected to reflect the actual execution load demanded from processors. Finally, Cache References and Cache

Misses amount provide a snapshot on memory access and memory prediction mechanisms. The ideal parameter to be measured would be memory usage, but it was not available to be retrieved on PERF tool and the dedicated tools for this kind of analysis are quite complex and conservative.

In each experiment, the application under evaluation was run 1000 times using the same set of inputs and results were averaged in order to provide adequate statistical comparison. Additional statistical data regarding maximum, minimum and standard deviation measurements are detailed in Chapter 5.

For all executions, the platforms had all network interfaces disabled in order to avoid any external interference from interrupts or polling mechanisms. Additionally, no other user applications were being executed in parallel.

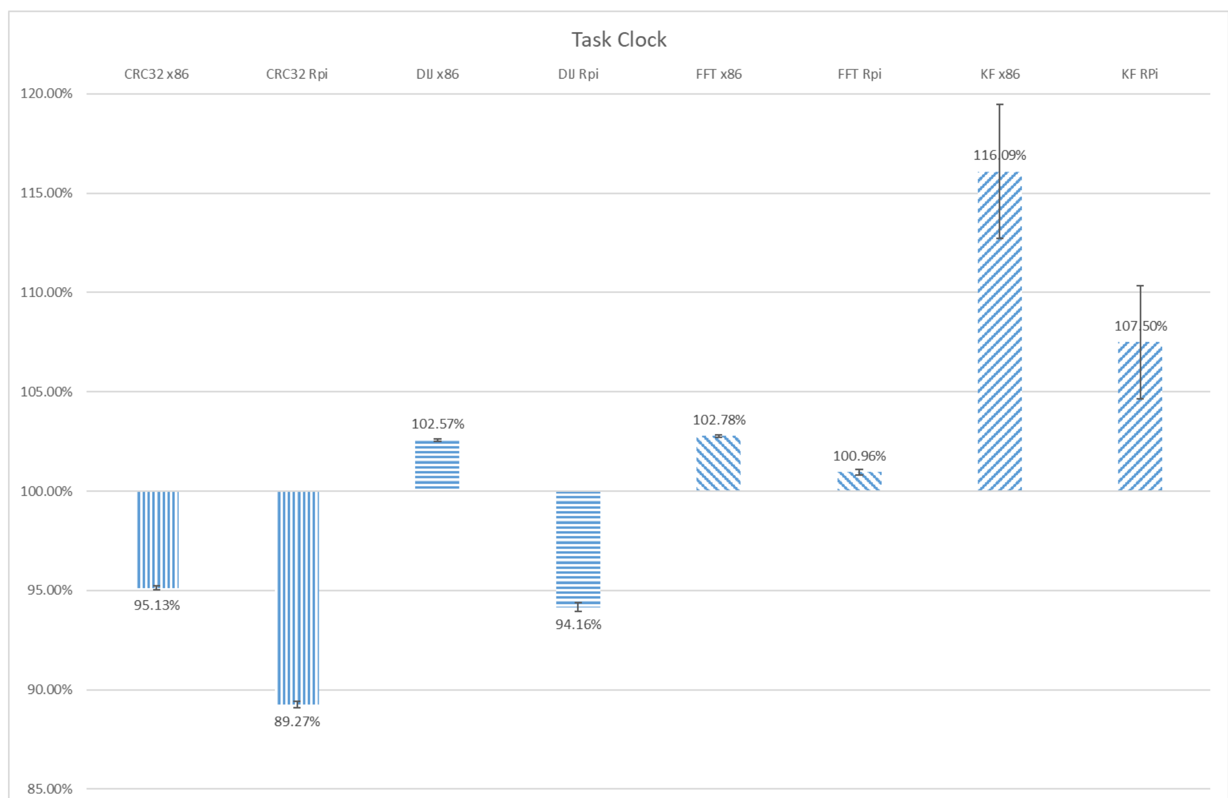
The obtained results are discussed in next chapter.

5 RESULTS AND DISCUSSION

Figure 5.1, Figure 5.2, Figure 5.3, Figure 5.4 and Figure 5.5 show the performance parameter results for each benchmark application running on both hardware platforms and considering implementation using FACE and non-FACE AIL versions.

Given the different nature of the chosen hardware platforms (x86 and Raspberry Pi), it is not the goal of this work to compare performance between them, since this comparison would not bring any contribution for the discussion proposed here. Considering this fact, the parameters displayed in graphs were normalized using non-FACE AIL execution as reference. It means each graph corresponds to a different parameter analyzed and each column shows the application parameter performance while running over a certain hardware platform. Therefore, results higher than 100% means a parameter result was higher in AIL FACE implementation than AIL Non-FACE implementation on the same hardware platform. The error bars on each columns correspond to the standard deviation of FACE implementation. The complete statistics for each parameter is presented in a table after the graph.

Figure 5.1 – Task Clock Time



Source: The Author (2019)

Task clock execution time parameter shown in Figure 5.1 presents different results among the four example benchmarks. FFT application had small overheads for FACE AIL implementation in both platforms: 2.78% while running over x86 and 0.96% while running over Raspberry Pi. CRC32 application had an improvement on task clock time on FACE AIL implementation over the x86 platform (4.87%) and a more significant one on the Raspberry Pi platform (11.83%), what may be a consequence of the usage of polynomial table allocated in global user memory. Dijkstra application had opposite results in the two hardware platforms, with a FACE AIL overhead of 2.57% for the x86 platform and an improvement of 6.84% for the Raspberry Pi platform. However, the more significant results were collected on Kalman Filter application, where overheads of 16.09% for the x86 platform and 7.50% for the Raspberry Pi platform were observed on FACE AIL implementation. This result could be consequence of the heavier iteration loops required by Kalman Filter algorithm that needs to execute several iterations on each execution in order to achieve the noise effect reduction on final solution and these iterations executed through FACE AIL layers may lead to this overhead.

The complete set of execution results is detailed on Table 5.1. The percentages in the table show the given parameter using the average value as reference, as detailed below:

- Average % (merged cell) = Average_FACE / Average_NonFACE;
- StdDev % (FACE) = StdDev_FACE / Average_FACE;
- Maximum % (FACE) = Max_FACE / Average_FACE;
- Minimum % (FACE) = Min_FACE / Average_FACE;
- StdDev % (Non-FACE) = StdDev_ Non-FACE / Average_ Non-FACE;
- Maximum % (Non-FACE) = Max_ Non-FACE / Average_ Non-FACE;
- Minimum % (Non-FACE) = Min_ Non-FACE / Average_ Non-FACE;

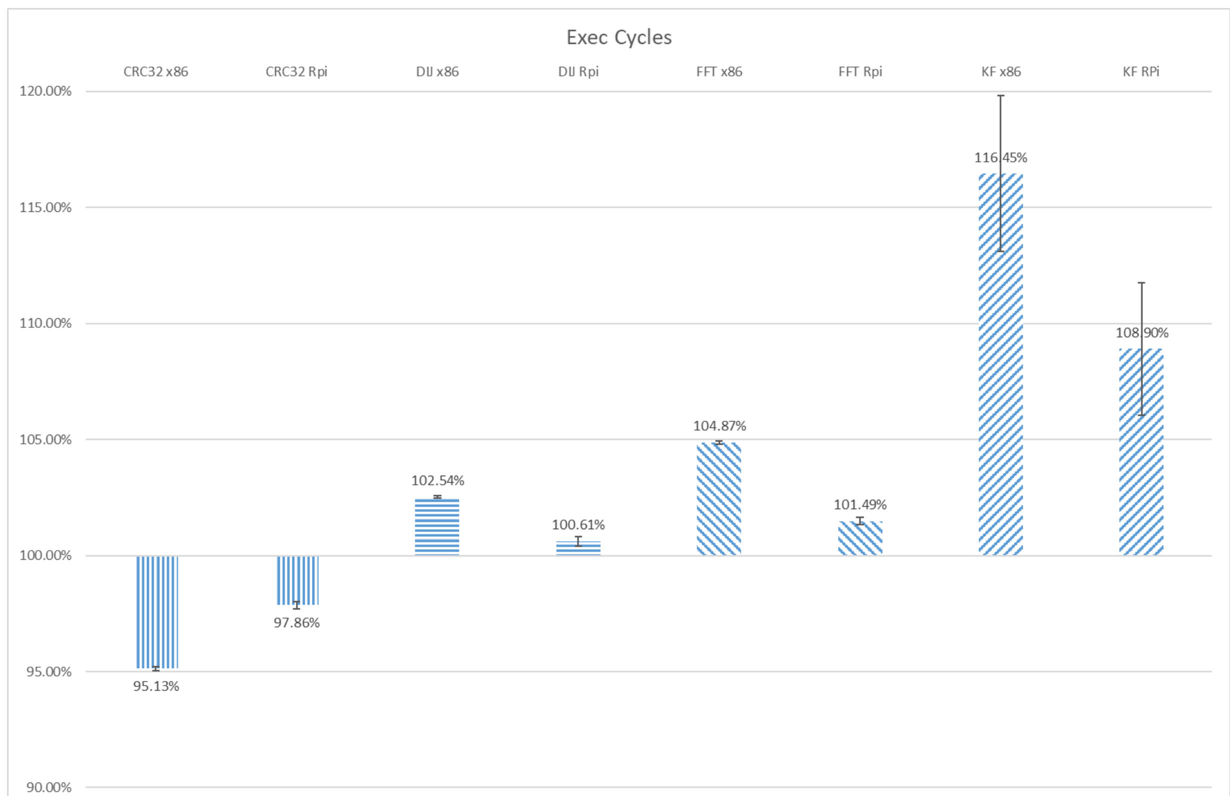
Table 5.1 – Complete Statistics for Task Clock Time.

Application	*Architecture*	*Average*	*Std Dev*	*Max*	*Min*
CRC32 x86	FACE	36151785.42	34154	36248659	36073182
	Non-FACE	36116769.82	33538	36206017	36036491
	FACE	100.10%	0.09%	100.27%	99.78%
	Non-FACE		0.09%	100.25%	99.78%
CRC32 RPi	FACE	49008383.51	75969	49483896	48881731
	Non-FACE	48527457.54	95947	48781355	48379790
	FACE	100.99%	0.16%	100.97%	99.74%
	Non-FACE		0.20%	100.52%	99.70%
DIJ x86	FACE	56510701.44	29901	56672080	56418119
	Non-FACE	55930642.11	29638	56063570	55837746
	FACE	101.04%	0.05%	100.29%	99.84%
	Non-FACE		0.05%	100.24%	99.83%
DIJ RPi	FACE	59873118.05	125130	60194350	59685627
	Non-FACE	59461043.65	127764	59799770	59254028
	FACE	100.69%	0.21%	100.54%	99.69%
	Non-FACE		0.21%	100.57%	99.65%
FFT x86	FACE	112068407.9	67057	112246593	111899694
	Non-FACE	108651894.9	66952	108821759	108486856
	FACE	103.14%	0.06%	100.16%	99.85%
	Non-FACE		0.06%	100.16%	99.85%
FFT RPi	FACE	108476076.9	153556	108841535	108233808
	Non-FACE	105859023.7	154553	106543742	105614172
	FACE	102.47%	0.14%	100.34%	99.78%
	Non-FACE		0.15%	100.65%	99.77%
KF x86	FACE	294287.683	9887	386353	289612
	Non-FACE	222227.244	8242	312973	217682
	FACE	132.43%	3.36%	131.28%	98.41%
	Non-FACE		3.71%	140.83%	97.95%
KF RPi	FACE	222034.044	6319	264668	214314
	Non-FACE	201129.346	7460	264847	193178
	FACE	110.39%	2.85%	119.20%	96.52%
	Non-FACE		3.71%	131.68%	96.05%

Source: Author (2019).

Results for processor execution cycles depicted in Figure 5.2 also demonstrated a wider difference while running Kalman Filter application compared to other application results. Unlike task clock parameter, execution cycles difference was more noticeable in the x86 platform (16.45% overhead on FACE AIL implementation) compared to the Raspberry Pi platform execution (8.90% overhead). This result may also be linked to the same hypothesis raised for task clock execution time, although for Raspberry Pi the impact was softer since the compiler in this platform may generate less instructions due to its simpler micro-architecture when compared to x86 platform. FFT application using FACE AIL consumed 4.87% more cycles than Non-FACE AIL version on x86 platform, while running on Raspberry Pi this difference was reduced to 1.49%. The same phenomenon was measured on Dijkstra application, despite the smaller results (FACE AIL usage overhead of 2.54% and 0.61% on the x86 and the Raspberry Pi, respectively). It is worth to notice that the opposite behavior was measured on CRC32 application, where FACE AIL usage actually brought a small performance improvement, taking less cycles to execute than in Non-FACE AIL version (4.87% improvement for x86 platform and 2.14% improvement on the Raspberry Pi). The complete set of execution results is detailed on Table 5.2.

Figure 5.2 – Task Executed Cycles



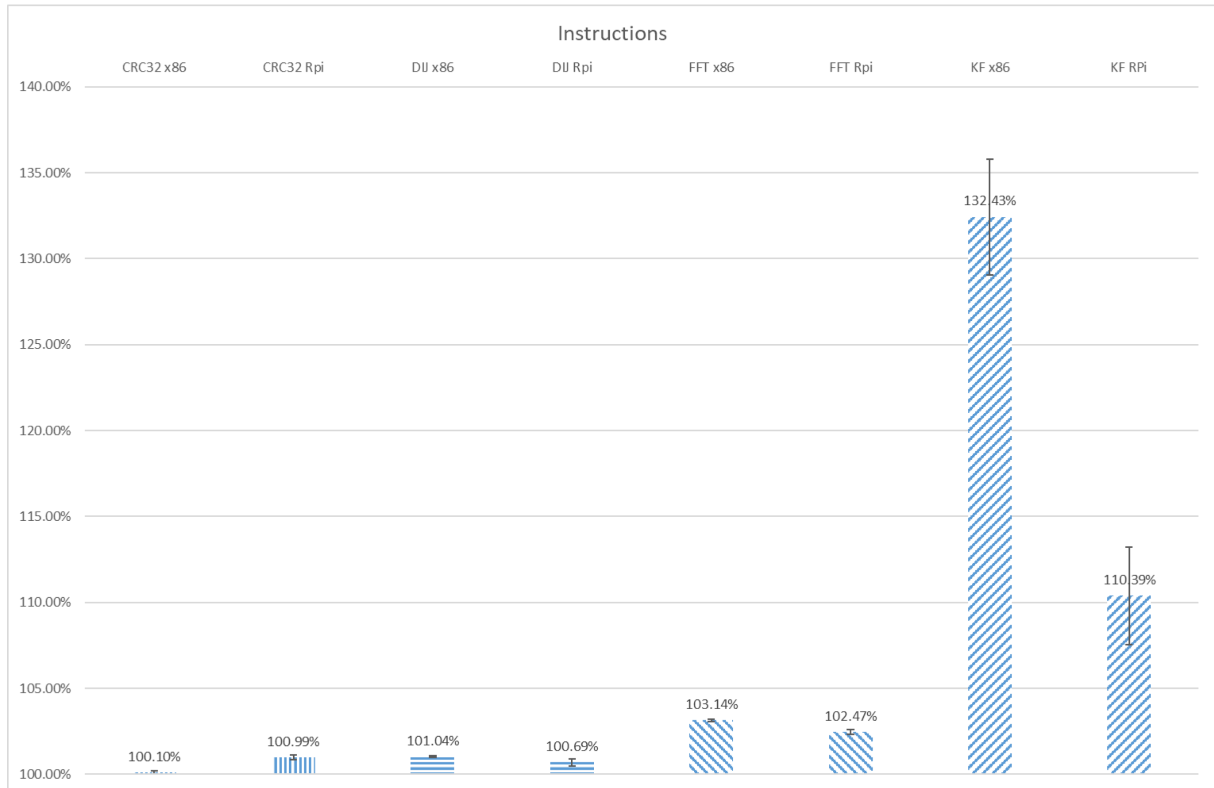
Source: The Author (2019)

Table 5.2 – Complete Statistics for Executed Cycles

Application	*Architecture*	*Average*	*Std Dev*	*Max*	*Min*
CRC32 x86	FACE	25022147.31	995164	32551382	24451963
	Non-FACE	26304223.2	1125943	35937788	24732554
	FACE	95.13%	3.98%	130.09%	97.72%
	Non-FACE		4.28%	136.62%	94.03%
CRC32 RPi	FACE	75660032.35	220937	77989509	74971451
	Non-FACE	77311003.15	279913	77843330	76684454
	FACE	97.86%	0.29%	103.08%	99.09%
	Non-FACE		0.36%	100.69%	99.19%
DIJ x86	FACE	29386147.81	2175552	41175421	28610179
	Non-FACE	28659207.79	1669357	40591757	28068548
	FACE	102.54%	7.40%	140.12%	97.36%
	Non-FACE		5.82%	141.64%	97.94%
DIJ RPi	FACE	95554571.26	394010	96664075	94889041
	Non-FACE	94971661.1	394593	95867739	94253437
	FACE	100.61%	0.41%	101.16%	99.30%
	Non-FACE		0.42%	100.94%	99.24%
FFT x86	FACE	85250774.67	4014179	104049906	83380605
	Non-FACE	81292208.83	3888874	99366615	79669159
	FACE	104.87%	4.71%	122.05%	97.81%
	Non-FACE		4.78%	122.23%	98.00%
FFT RPi	FACE	182801647.8	661566	184511221	181501183
	Non-FACE	180120291.6	794143	185469731	178417029
	FACE	101.49%	0.36%	100.94%	99.29%
	Non-FACE		0.44%	102.97%	99.05%
KF x86	FACE	406725.94	32339	692591	389528
	Non-FACE	349279.687	26399	591098	335223
	FACE	116.45%	7.95%	170.28%	95.77%
	Non-FACE		7.56%	169.23%	95.98%
KF RPi	FACE	673788.806	52526	882361	566850
	Non-FACE	618731.612	56651	916857	516198
	FACE	108.90%	7.80%	130.96%	84.13%
	Non-FACE		9.16%	148.18%	83.43%

Source: Author (2019).

Figure 5.3 – Executed Instructions



Source: The Author (2019)

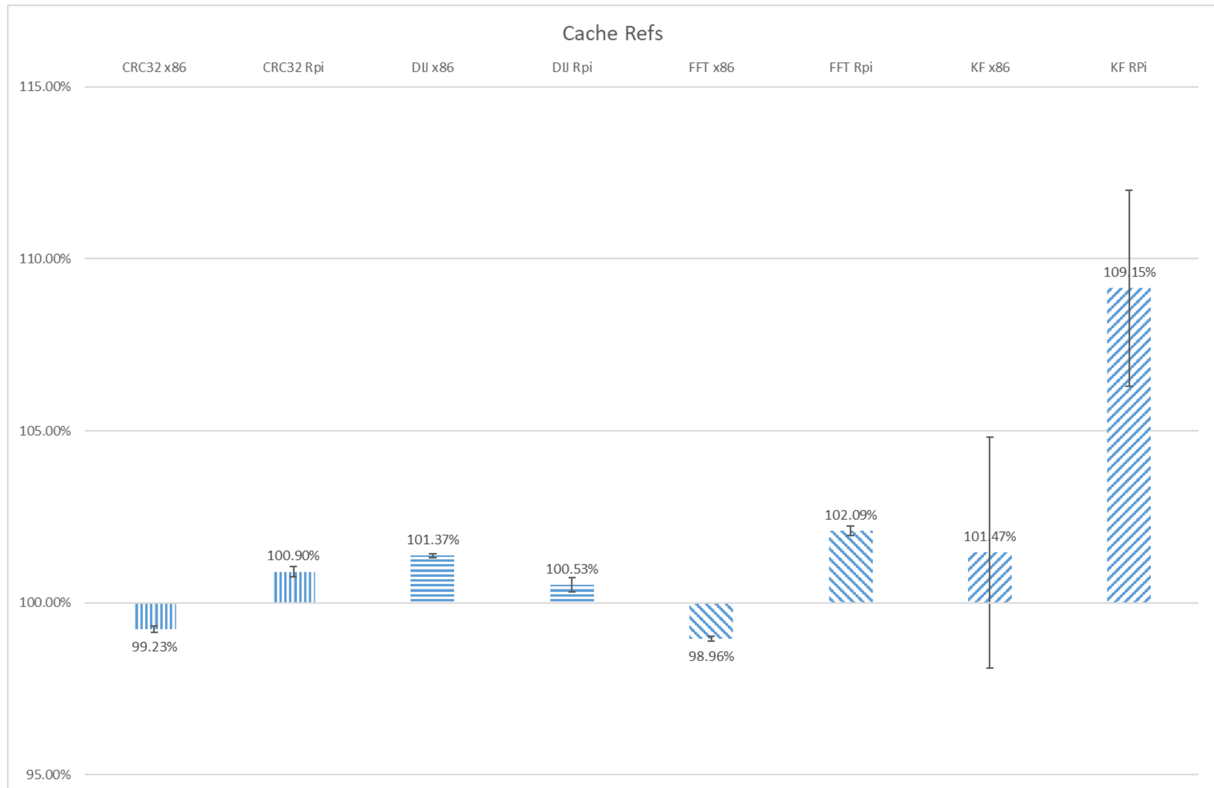
The number of executed instructions parameter presented in Figure 5.3 depicts a similar overhead for FFT application on FACE AIL implementation for both x86 platform (3.14% increase) and Raspberry Pi platform (2.47% increase). The same effect was observed for Dijkstra application implemented with FACE AIL but with reduced impact (1.04% overhead running over the x86 and 0.69% overhead running over the Raspberry Pi). A small, but yet noticeable, difference was observed on CRC32 application, where there was also an overhead on FACE AIL implementation (0.10% and 0.99% overhead for the x86 and the Raspberry Pi platforms, respectively). The more significant gap was again on Kalman Filter application, which presented an overhead of 32.42% on FACE AIL implementation running over the x86 platform. Its Raspberry Pi execution also had an overhead, although smaller (10.39%), which is coherent with execution cycles result. The complete set of execution results is detailed on Table 5.3.

Table 5.3 – Complete Statistics for Executed Instructions.

Application	*Architecture*	*Average*	*Std Dev*	*Max*	*Min*
CRC32 x86	FACE	36151785.42	34154	36248659	36073182
	Non-FACE	36116769.82	33538	36206017	36036491
	FACE	100.10%	0.09%	100.27%	99.78%
	Non-FACE		0.09%	100.25%	99.78%
CRC32 RPi	FACE	49008383.51	75969	49483896	48881731
	Non-FACE	48527457.54	95947	48781355	48379790
	FACE	100.99%	0.16%	100.97%	99.74%
	Non-FACE		0.20%	100.52%	99.70%
DIJ x86	FACE	56510701.44	29901	56672080	56418119
	Non-FACE	55930642.11	29638	56063570	55837746
	FACE	101.04%	0.05%	100.29%	99.84%
	Non-FACE		0.05%	100.24%	99.83%
DIJ RPi	FACE	59873118.05	125130	60194350	59685627
	Non-FACE	59461043.65	127764	59799770	59254028
	FACE	100.69%	0.21%	100.54%	99.69%
	Non-FACE		0.21%	100.57%	99.65%
FFT x86	FACE	112068407.9	67057	112246593	111899694
	Non-FACE	108651894.9	66952	108821759	108486856
	FACE	103.14%	0.06%	100.16%	99.85%
	Non-FACE		0.06%	100.16%	99.85%
FFT RPi	FACE	108476076.9	153556	108841535	108233808
	Non-FACE	105859023.7	154553	106543742	105614172
	FACE	102.47%	0.14%	100.34%	99.78%
	Non-FACE		0.15%	100.65%	99.77%
KF x86	FACE	294287.683	9887	386353	289612
	Non-FACE	222227.244	8242	312973	217682
	FACE	132.43%	3.36%	131.28%	98.41%
	Non-FACE		3.71%	140.83%	97.95%
KF RPi	FACE	222034.044	6319	264668	214314
	Non-FACE	201129.346	7460	264847	193178
	FACE	110.39%	2.85%	119.20%	96.52%
	Non-FACE		3.71%	131.68%	96.05%

Source: Author (2019).

Figure 5.4 – Cache References



Source: The Author (2019)

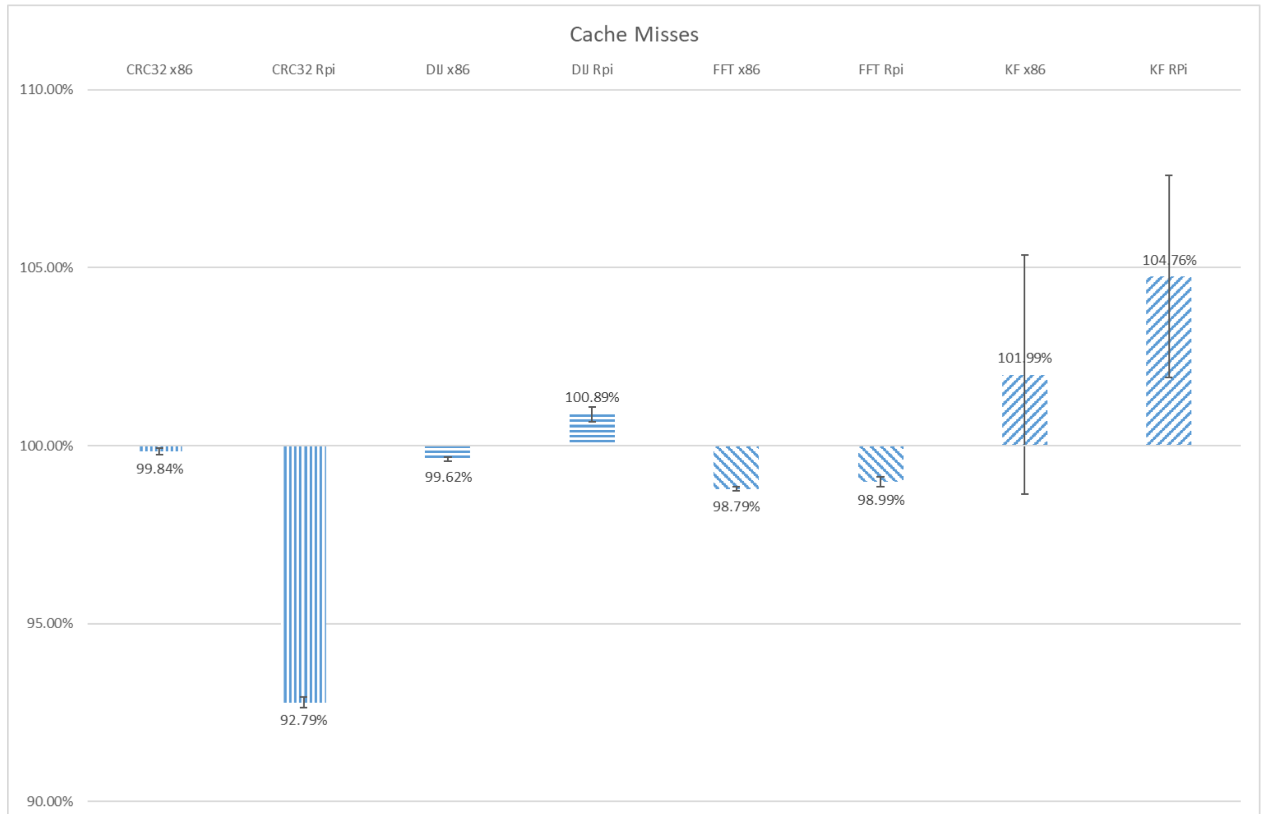
Cache Reference measurements (Figure 5.4) on FFT application had a small reduction on FACE AIL implementation running on x86 (1.04%) and a small increase on Raspberry Pi (2.09%). For Dijkstra application the measured overhead was also small. However, it had differences between the x86 execution, where the overhead was 1.37% for the x86 platform, and the Raspberry Pi execution that presented 0.53% overhead for FACE AIL implementation. Kalman Filter application had more relevant overheads of 1.47% for the x86 platform and 9.15% for the Raspberry Pi platform, where such difference for the impact on Raspberry Pi could be linked to the more reduce availability of main memory on this platform. Finally, CRC32 application presented small differences, but in the opposite direction: a FACE AIL implementation overhead of 0.90% on the Raspberry Pi execution and a marginal improvement of 0.77% on FACE AIL running over the x86 platform. The complete set of execution results is detailed on Table 5.4.

Table 5.4 – Complete Statistics for Cache References.

Application	*Architecture*	*Average*	*Std Dev*	*Max*	*Min*
CRC32 x86	FACE	16684	929	19073	13311
	Non-FACE	16814	943	21554	13386
	FACE	99.23%	5.57%	114.32%	79.78%
	Non-FACE		5.61%	128.19%	79.61%
CRC32 RPi	FACE	23233345	64726	23583285	23094965
	Non-FACE	23025478	87574	23314822	22864537
	FACE	100.90%	0.28%	101.51%	99.40%
	Non-FACE		0.38%	101.26%	99.30%
DIJ x86	FACE	13699	1135	19745	11472
	Non-FACE	13514	1034	19696	11665
	FACE	101.37%	8.28%	144.13%	83.74%
	Non-FACE		7.65%	145.74%	86.32%
DIJ RPi	FACE	38462619	58036	38636603	38374152
	Non-FACE	38260411	59945	38441185	38161869
	FACE	100.53%	0.15%	100.45%	99.77%
	Non-FACE		0.16%	100.47%	99.74%
FFT x86	FACE	66932	5550	98379	51256
	Non-FACE	67637	5314	82599	49068
	FACE	98.96%	8.29%	146.98%	76.58%
	Non-FACE		7.86%	122.12%	72.55%
FFT RPi	FACE	51102880	70392	51290164	50993856
	Non-FACE	50055907	70577	50382549	49944644
	FACE	102.09%	0.14%	100.37%	99.79%
	Non-FACE		0.14%	100.65%	99.78%
KF x86	FACE	5861	197	7683	5455
	Non-FACE	5776	210	7349	5417
	FACE	101.47%	3.36%	131.10%	93.08%
	Non-FACE		3.64%	127.24%	93.79%
KF RPi	FACE	94486	2713	113809	90736
	Non-FACE	86569	3357	116288	82718
	FACE	109.15%	2.87%	120.45%	96.03%
	Non-FACE		3.88%	134.33%	95.55%

Source: Author (2019).

Figure 5.5 – Cache Misses



Source: The Author (2019)

Finally, Cache Misses results (Figure 5.5) showed distinct results compared to Cache References. For FFT application, FACE AIL implementation had a slight improvement compared to Non-FACE AIL of 1.21% on the x86 platform and 1.01% on the Raspberry Pi platform. A similar result was observed also in CRC32 application, however with a more considerable improvement in favor of FACE AIL experimented on the Raspberry Pi platform (7.21%) rather than the slight improvement on x86 platform (0.26%). Despite Kalman Filter application had a negligible overhead on FACE AIL implementation over the x86 platform (1.99%), while executed over the Raspberry Pi such overhead was more significant (4.76%). Dijkstra application presented a small performance improvement with FACE AIL of 0.38% on the x86 platform and a small overhead of 0.89% on the Raspberry Pi. In a general way, besides the improvement observed on CRC32 running over Raspberry Pi, no considerable impact was measured on cache misses. The complete set of execution results is detailed on Table 5.5.

Table 5.5 – Complete Statistics for Cache Misses.

Application	*Architecture*	*Average*	*Std Dev*	*Max*	*Min*
CRC32 x86	FACE	16684	929	19073	13311
	Non-FACE	16814	943	21554	13386
	FACE	99.23%	5.57%	114.32%	79.78%
	Non-FACE		5.61%	128.19%	79.61%
CRC32 RPi	FACE	23233345	64726	23583285	23094965
	Non-FACE	23025478	87574	23314822	22864537
	FACE	100.90%	0.28%	101.51%	99.40%
	Non-FACE		0.38%	101.26%	99.30%
DIJ x86	FACE	13699	1135	19745	11472
	Non-FACE	13514	1034	19696	11665
	FACE	101.37%	8.28%	144.13%	83.74%
	Non-FACE		7.65%	145.74%	86.32%
DIJ RPi	FACE	38462619	58036	38636603	38374152
	Non-FACE	38260411	59945	38441185	38161869
	FACE	100.53%	0.15%	100.45%	99.77%
	Non-FACE		0.16%	100.47%	99.74%
FFT x86	FACE	66932	5550	98379	51256
	Non-FACE	67637	5314	82599	49068
	FACE	98.96%	8.29%	146.98%	76.58%
	Non-FACE		7.86%	122.12%	72.55%
FFT RPi	FACE	51102880	70392	51290164	50993856
	Non-FACE	50055907	70577	50382549	49944644
	FACE	102.09%	0.14%	100.37%	99.79%
	Non-FACE		0.14%	100.65%	99.78%
KF x86	FACE	5861	197	7683	5455
	Non-FACE	5776	210	7349	5417
	FACE	101.47%	3.36%	131.10%	93.08%
	Non-FACE		3.64%	127.24%	93.79%
KF RPi	FACE	94486	2713	113809	90736
	Non-FACE	86569	3357	116288	82718
	FACE	109.15%	2.87%	120.45%	96.03%
	Non-FACE		3.88%	134.33%	95.55%

Source: Author (2019).

From the data obtained from experiments conducted using four different benchmark applications and also over two different hardware platforms, it was possible to observe that the implementation of abstraction layers required by Open Systems Architectures (FACE, in this case study) did not bring major overheads in the five performance parameters analyzed. Moreover, in some scenarios the implementation of FACE even brought performance improvements when compared to non-modular library implementation. Eventually it could be observed that task clock and execution cycles parameters were strongly linked. The same issue happened also between cache references and cache misses. For a new experiment it

would be interesting to verify more dissociated parameters. Additionally, a considerable standard deviation was observed in all KF measurements, especially on cache references and cache misses, for both in x86 and RPi. This phenomenon may be related to the smaller readouts provided by this application when compared to other applications, and therefore small acquisition fluctuations would have a higher impact on standard deviation.

However, since it was not possible to model a consistent behavior among all benchmark application examples, when comparing implementations using a traditional architecture to an open system architecture, it is recommended to prototype the implementation beforehand and evaluate key performance indicators relevant to the application under development, especially when platforms with limited hardware resources are used. This outcome is similar to what was observed in the related works regarding manual optimizations that were still required after adoption of portable code for application running over GPUs in order to maintain performance while porting an application between different platforms (i.e.: performance portability).

The prototyping concept is also applicable if a new application is being developed and there is freedom to decide which Open Systems Architecture will be adopted. If the decision by a certain OSA type is already placed as a customer or internal requirement, it is still recommended to prototype and evaluate key features for the application, in order to prioritize design and implementation decisions to minimize overheads or even further explore benefits due to Open Systems Architecture adoption such as the performance improvements observed in the experiments.

6 CONCLUSION

The increased reuse and modularity as a consequence of an architecture based on well-defined abstraction layers, leading to a reduction on development costs and time-to-field are benefits from Future Airborne Capability Environment (FACE) as an Open Systems Architecture example of usage in avionics embedded systems. However, one of the main question marks, based on the experience on other domains, was the overhead that FACE architecture might insert due to its standardization layers.

This work focused on addressing this concern, verifying the existence and quantifying such impact based on the analysis of five different key performance parameters while using a library implementing FACE architecture as a case study and running four benchmark applications on top of it. The same experiment was conducted using a library version with a non-modular implementation (i.e.: without adherence to FACE Architecture) in order to have a comparison baseline. Additionally, this quantification process was conducted over two different hardware platforms widely used in embedded systems (x86 and Raspberry Pi).

The obtained results demonstrated that in most cases there is no significant impact caused by the implementation of abstraction layers required by FACE Technical Standard concerning the five performance parameters analyzed when compared to a non-modular implementation that provides the same functionalities. In most cases the impact of such overheads varied from virtually zero up to 5% overhead, depending on the nature of application and the hardware platform used. In certain scenarios, even a performance improvement was observed on implementation using FACE architecture compared to traditional, non-modular implementation.

Nevertheless, a general modeling for such performance differences could not be achieved based on data obtained during the experiments and therefore prototyping and performing quantitative assessments are still recommended as the best option to evaluate open systems architecture implementation impact on key performance parameters, especially on hardware platforms with limited resources. Considering the constraints of each application and platform it will be possible to take the best design decisions in order to leverage from modularity and ease of reuse offered by FACE Architecture, while respecting performance constraints that will assure that a given application will accomplish its mission.

Based on these results, this work conclusion is that the penalties measured on system performance were not high enough to jeopardize performance and the benefits brought on modularity and reuse brought by FACE made it worth its adoption

Future works intend to further explore these findings on more complex software application types, namely using end-to-end avionics applications and verifying other performance parameters regarding their variability and diversity to explore more characteristics in order to evaluate and propose improvements to minimize or avoid overheads and therefore benefit from Open Systems Architecture abstraction and modularity without jeopardizing system key parameters performance.

Another field for further works exploration is to focus the performance analysis while using safety-critical profile, which inserts more constraints to software developers and system integrators given the reduced options of POSIX calls available and few options for real-time operating systems choice. This would be beneficial since a considerable part of defense applications are under this FACE profile, given the nature and criticality of their missions. Finally, it can be explored the possibility or running the same kind of test using a processor simulator like gem5, where it is possible to have access and monitor the complete set of micro-architectural counters, retrieving more accurate and insightful measurements.

REFERENCES

ARINC. **ARINC Specification 653: Part 1, Avionics Application Software Standard Interface, Required Services**. 2006.

BISHOP, G. et al. An introduction to the Kalman Filter. **Proc of SIGGRAPH, Course,** v. 8, n. 27599-23175, p. 41, 2001.

BRABSON, S.; ANDERSON, T. Evolution of the US Navy's Collision Avoidance Systems (CAS) to Future Airborne Capability Environment (FACE). In: **IEEE A&E SYSTEMS MAGAZINE**, June 2015.

BLOOM, G.; SHERRILL, J.; GILLILAND, G. Aligning Deos and RTEMS with the FACE safety base operating system profile. **ACM Sigbed Review**, [s.l.], v. 15, n. 1, p.15-21, 20 mar. 2018. Association for Computing Machinery (ACM). http://dx.doi.org/10.1145/3199610.3199612.

DIMAKOPOULOU, M. et al. Reliable and efficient performance monitoring in linux. In: **Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis**. IEEE Press, 2016. p. 34.

DRAGAN, L.; WATT, S. M. Performance analysis of generics in scientific computing. In: SEVENTH INTERNATIONAL SYMPOSIUM ON SYMBOLIC AND NUMERIC ALGORITHMS FOR SCIENTIFIC COMPUTING (SYNASC'05), 7., 2005, Timisoara, Romania. **Proceedings of Seventh International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC'05).** Timisoara, Romania: IEEE, 2005. p. 1 - 8.

EXÉRCITO BRASILEIRO. **Sistema Defesa, Indústria e Academia (SisDIA) de Inovação**. Retrieved 11 July 19. Available from Internet: <http://www.dct.eb.mil.br/index.php/sistema-defesa-industria-e-academia>.

ETZKOWITZ, H.; CHUNYAN Z. **The triple helix: University–industry–government innovation and entrepreneurship.** Routledge, 2017.

FABEIRO, J. F. et al. Automatic Generation of Optimized OpenCL Codes Using OCLoptimizer, **The Computer Journal**, Volume 58, Issue 11, November 2015, Pages 3057–3073, https://doi.org/10.1093/comjnl/bxv038

FEDERAL AVIATION ADMINISTRATION. **FAA Order 8040.4B Safety Risk Management Policy Document Information**. Issued on May 02, 2017. Retrieved 11 July, 2019. Available from Internet: <https://www.faa.gov/regulations_policies/orders_notices/index.cfm/go/document.current/documentNumber/8040.4>.

GALSTER, M.; EBERLEIN, A.; MOUSSAVI, M. Systematic selection of software architecture styles. **IET Software**, [s.l.], v. 4, n. 5, p.349-360, 2010. Institution of Engineering and Technology (IET). http://dx.doi.org/10.1049/iet-sen.2009.0004.

GUTHAUS, M. R. et al. MiBench: A free, commercially representative embedded benchmark suite. **Proceedings of the Fourth Annual IEEE International Workshop On Workload Characterization. Wwc-4 (cat. No.01ex538)**, [s.l.], 2001. IEEE. p.1-12. http://dx.doi.org/10.1109/wwc.2001.990739

HAN, S.; HYUN-WOOK J. Resource partitioning for Integrated Modular Avionics: comparative study of implementation alternatives. **Software: Practice and Experience** 44, no. 12 (2014): p. 1441-1466.

IEEE STANDARDS ASSOCIATION. IEEE 1003.1-2017: IEEE Standard for Information Technology--Portable Operating System Interface (POSIX(R)) Base Specifications. 7 ed. USA: IEEE, 2017.

KOMATSU, K. et al. Evaluating Performance and Portability of OpenCL Programs. In: THE FIFTH INTERNATIONAL WORKSHOP ON AUTOMATIC PERFORMANCE TUNING, 2010, Berkeley, CA, USA. **Proceedings of 5th International Workshop on Automatic Performance Tuning.** Berkeley, CA, USA: Iwapt2010, 2010. p. 1 - 15.

LEWIS, J.; RIERSON, L. Certification concerns with integrated modular avionics (IMA) projects. In: THE 22nd DIGITAL AVIONICS SYSTEMS CONFERENCE, 2003. DASC '03, 22, 2003, Indianapolis, IN, USA**. Proceedings of 22nd Digital Avionics Systems Conference, 2003**. Indianapolis, IN, USA: IEEE, 2003. p. 1 - 9.

LÖFWENMARK, A.; NADJM-TEHRANI, S. Challenges in future avionic systems on multi-core platforms. In **2014 IEEE International Symposium on Software Reliability Engineering Workshops**, Naples, Italy, IEEE, 2014. p. 115-119.

MCHALE, J. **PC/104 and small form factors popular in defense electronics systems.** Retrieved 11 July 2019. Available from Internet: <http://mil-embedded.com/articles/pc104-small-popular-defense-electronics-systems/>.

POP, P., et al. Methods and tools for reducing certification costs of mixed-criticality applications on multi-core platforms: the RECOMP approach. In: **Proceedings of the Workshop of Industry-Driven Approaches for Cost-effective Certification of Safety-Critical, Mixed-Criticality Systems**, vol. 156. 2013.

PRISAZNUK, P. J. ARINC 653 role in integrated modular avionics (IMA). In: 2008 IEEE/AIAA 27TH DIGITAL AVIONICS SYSTEMS CONFERENCE, 2008, St. Paul, MN, USA. **Proceedings of 27th Digital Avionics Systems Conference, 2008**, St. Paul, MN, USA: IEEE, 2008, p. 1-E.

RASPBERRY PI. **Raspberry Pi 3 Model B.** Retrieved 11 July, 2019. Available from Internet: <https://www.raspberrypi.org/products/raspberry-pi-3-model-b/>

RTCA. DO-178B/ED-12B: **Software Considerations in Airborne Systems and Equipment Certification**. RTCA Inc., 1992.

RTCA. DO-297: **Integrated Modular Avionics (IMA) Development Guidance and Certification Considerations**. RTCA Inc., 2005.

RUAN, W.; ZHENGJUN Z. Kernel-level design to support partitioning and hierarchical real-time scheduling of ARINC 653 for VxWorks. In: **2014 IEEE 12th International Conference on Dependable, Autonomic and Secure Computing**, Dalian, China. IEEE, 2014. pp. 388-393.

RUL, S. et al. An experimental study on performance portability of OpenCL kernels. In: **Application Accelerators In High Performance Computing, 2010 Symposium, Papers**, 2010, Knoxville, TN, USA. Papers. Knoxville, TN, USA: UGent, 2010. p. 1 - 3. Disponível em: <http://hdl.handle.net/1854/LU-1016024>. Acesso em: 11 jul. 2019

SAE INTERNATIONAL. AS6512: Unmanned Systems (UxS) Control Segment (UCS) Architecture: Architecture Description. Warrendale, PA, USA: SAE International, 2016.

SANTOS, N. et al. Development of a software platform to control squads of unmanned vehicles in real-time. In: **2017 International Conference on Unmanned Aircraft Systems (ICUAS)**. IEEE, 2017. p. 1-5.

SIPRI. **SIPRI Military Expenditure Database**. Retrieved 11 July 2019. Available from Internet: <https://www.sipri.org/sites/default/files/SIPRI-Milex-data-1949-2018_0.xlsx>.

SHAFER, J.; RIXNER, S.; COX, A. L. The Hadoop distributed filesystem: Balancing portability and performance. In: 2010 IEEE INTERNATIONAL SYMPOSIUM ON PERFORMANCE ANALYSIS OF SYSTEMS & SOFTWARE (ISPASS), 2010, 2010, White Plains, NY, USA. **Proceedings of 2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS).** White Plains, NY, USA: IEEE, 2010. p. 122 - 133.

THE OPEN GROUP (Ed.). Future Airborne Capability Environment Consortium. **Future Airborne Capability Environment (FACE) Overview**. USA: The Open Group, 2016. Retrieved 11 July, 2019. Available from Internet: <https://publications.opengroup.org/guides/face/g165>.

THE OPEN GROUP (Ed.). Future Airborne Capability Environment Consortium. **FACE Technical Standard 3.0.** ed. USA: The Open Group, 2017. Retrieved 11 July, 2019. Available from Internet: <https://publications.opengroup.org/c17c>.

TOKAR, J. L. An Examination of Open System Architectures for Avionics Systems – An Update. In: THE U.S. AIR FORCE FACE TECHNICAL INTERCHANGE MEETING, 2017., 2017, Dayton, OH, USA. **The U.S. Air Force FACE™ Technical Interchange Meeting Summary**. Dayton, OH, USA: The Open Group, 2017. p. 1 - 21. Available from Internet:
https://www.opengroup.us/face/documents/18456/Pyrrhus_Examination_of_Open_Systems_architecture_Paper.pdf

UNITED STATES OF AMERICA. Office Of The Under Secretary Of Defense For Acquisition & Sustainment. Department Of Defense. **Implementation Directive for Better Buying Power 3.0 – Achieving Dominant Capabilities through Technical Excellence and Innovation**. 2015. Retrieved 11 July, 2019. Available from Internet:
<https://www.acq.osd.mil/fo/docs/betterBuyingPower3.0(9Apr15).pdf>.

UNITED STATES OF AMERICA. Defense Acquisition University. Department Of Defense. **Defense Acquisition Guidebook**. 2017. Retrieved 11 July, 2019. Available from Internet:: <https://www.dau.mil/tools/dag>.

UNITED STATES OF AMERICA. Department of Defense. **DoDI 5000.02 Operation of the Defense Acquisition System**. 2017. D Retrieved 11 July, 2019. Available from Internet: <http://www.esd.whs.mil/Portals/54/Documents/DD/issuances/dodi/500002_dodi_2015.pdf>.

UNITED STATES OF AMERICA. Department of Defense. **MIL-STD-498, Military Standard: Software Development and Documentation**. 1994. Retrieved 11 July, 2019. Available from Internet: <http://everyspec.com/MIL-STD/MIL-STD-0300-0499/MIL-STD-498_25500/>.

VARBANESCU, A. L. et al. Can Portability Improve Performance? **Proceedings of the 6th ACM/Spec International Conference on Performance Engineering - ICPE '15**, [s.l.], p.277-287, 2015. ACM Press. http://dx.doi.org/10.1145/2668930.2688042.

VR ASSETS. **VPC-SA31FX Motherboard 4GB Intel i5-2430M 2.4 MBX-237 A1846546A**. Retrieved 11 July 2019. Available from Internet: <https://www.vrassets.us/sony-vaio-vpc-sa31fx-motherboard-4gb-intel-i5-2430m-2-4-mbx-237-a1846546a-as-is.html>.

WALLACE, J. et al. Resilient-Embedded Global Positioning System/Inertial Navigation System (R-EGI) virtual System Integration Laboratory (vSIL) Prototyping Initiative Phase 1 and Phase 2. In: MARCH 2017 FACE AIR FORCE TIM PAPER, 2017, 2017, Dayton, OH, USA. **Proceedings of March 2017 FACE Air Force TIM**. Dayton, OH, USA: The Open Group, 2017. p. 1 - 14.

WATKINS, C. B.; WALTER, R. Transitioning from federated avionics architectures to Integrated Modular Avionics. In: 2007 IEEE/AIAA 26TH DIGITAL AVIONICS SYSTEMS CONFERENCE, 2007, Dallas, TX, USA. **Proceedings of 2007 IEEE/AIAA 26th Digital Avionics Systems Conference**. Dallas, TX, USA: IEEE, 2007. p. 1 - 10.

WILHELM, R. et al. The worst-case execution-time problem—overview of methods and survey of tools. **ACM Transactions on Embedded Computing Systems (TECS)**, v. 7, n. 3, p. 36, 2008.

WOLFIG, R.; MIRKO J. Distributed IMA and DO-297: Architectural, communication and certification attributes. In: **2008 IEEE/AIAA 27th Digital Avionics Systems Conference**, St. Paul, MN, USA. IEEE, 2008. pp. 1-E.

YU, J. et al. Radon-Fourier transform for radar target detection (III): Optimality and fast implementations. **IEEE Transactions on Aerospace and Electronic Systems**, v. 48, n. 2, p. 991-1004, 2012.