Application des architectures many core dans les systèmes embarqués temps réel

Moustapha Lo

Communauté
UNIVERSITÉ Grenoble Alpes

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Informatique**

Arrêté ministériel : 25 Mai 2016

Présentée par

Moustapha Lô

Thèse dirigée par **Florence MARANINCHI**
et codirigée par **Pascal RAYMOND**

préparée au sein de **Verimag** et **Airbus Helicopters**
et de l'**École Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique**

Implementing a Real-time Avionic application on a Many-core Processor

Thèse soutenue publiquement **le 22 Fevrier 2019** ,
devant le jury composé de :

Claire Pagetti
HDR, Ingénieur de recherche, ONERA, Rapporteur
Emmanuel Grolleau
Professeur, ENSMA, Président
Giuseppe Lipari
Professeur, Université de Lille, Examinateur
Florence MARANINCHI
Professeur, Grenoble INP, Directrice de thèse
Pascal RAYMOND
Chargé de recherche, CNRS, Co-encadrant
Nicolas VALOT
Expert Logiciel Embarqué, Airbus Helicopters, Examinateur

To my mother Anna Bâ for her unconditional love and support. Without you, I would not be who I am today.

Remerciements

Un GRAND MERCI à tous et à bientôt ☺
Moustapha

Résumé

Les processeurs mono-coeurs traditionnels ne sont plus suffisants pour répondre aux besoins croissants en performance des fonctions avioniques. Les processeurs multi/many-coeurs ont emergé ces dernières années afin de pouvoir intégrer plusieurs fonctions et de bénéficier de la puissance par Watt disponible grâce aux partages de ressources. En revanche, tous les processeurs multi/many-coeurs ne répondent pas forcément aux besoins des fonctions avioniques. Nous préférons avoir plus de déterminisme que de puissance de calcul car la certificabilité de ces processeurs passe par la maîtrise du déterminisme.

L'objectif de cette thèse est d'évaluer le processeur many-coeur (MPPA-256) de Kalray dans un contexte industriel aéronautique. Nous avons choisi la fonction de maintenance HMS (Health Monitoring System) qui a un besoin important en bande passante et un besoin de temps de réponse borné. Par ailleurs, cette fonction est également dotée de propriétés de parallélisme car elle traite des données de vibration venant de capteurs qui sont fonctionnellement indépendants, et par conséquent leur traitement peut être parallélisé sur plusieurs coeurs. La particularité de cette étude est qu'elle s'intéresse au déploiement d'une fonction existante séquentielle sur une architecture many-coeurs en partant de l'acquisition des données jusqu'aux calculs des indicateurs de santé avec un fort accent sur le flux d'entrées/sorties des données. Nos travaux de recherche ont conduit à 5 contributions:

- *Transformation des algorithmes existants en algorithmes incrémentaux capables de traiter les données au fur et mesure qu'elles arrivent des capteurs.*
- *Gestion du flux d'entrée des échantillons de vibrations jusqu'aux calculs des indicateurs de santé, la disponiblité des données dans le cluster interne, le moment où elles sont consommées et enfin l'estimation de la charge de calcul.*
- *Mesures de temps pas très intrusives directement sur le MPPA-256 en ajoutant des timestamps dans le flot de données.*
- *Architecture logicielle qui respecte les contraintes temps-réel même dans les pires cas. Elle est basée sur une pipeline à 3 étages.*
- *Illustration des limites de la fonction existante: nos expériences ont montré que les paramètres contextuels de l'hélicoptère tels que la vitesse du rotor doivent être corrélés aux indicateurs de santé pour réduire les fausses alertes.*

Mots-clés: Many-core processor, HMS (Health Monitoring System), Déterminisme, Temps-réel, Parallélisme, Transformations d'algorithmes, Algorithmes globaux, Algorithmes incrémentaux.

Abstract

Traditional single-cores are no longer sufficient to meet the growing needs of performance in avionics domain. Multi-core and many-core processors have emerged in the recent years in order to integrate several functions thanks to the resource sharing. In contrast, all multi-core and many-core processors do not necessarily satisfy the avionic constraints. We prefer to have more determinism than computing power because the certification of such processors depends on mastering the determinism.

The aim of this thesis is to evaluate the many-core processor (MPPA-256) from Kalray in avionic context. We choose the maintenance function HMS (Health Monitoring System) which requires an important bandwidth and a response time guarantee. In addition, this function has also parallelism properties. It computes data from sensors that are functionally independent and, therefore their processing can be parallelized in several cores. This study focuses on deploying the existing sequential HMS on a many-core processor from the data acquisition to the computation of the health indicators with a strong emphasis on the input flow. Our research led to five main contributions:

- *Transformation of the global existing algorithms into a real-time ones which can process data as soon as they are available.*
- *Management of the input flow of vibration samples from the sensors to the computation of the health indicators, the availability of raw vibration data in the internal cluster, when they are consumed and finally the workload estimation.*
- *Implementing a lightweight Timing measurements directly on the MPPA-256 by adding timestamps in the data flow.*
- *Software architecture that respects real-time constraints even in the worst cases. The software architecture is based on three pipeline stages.*
- *Illustration of the limits of the existing function: our experiments have shown that the contextual parameters of the helicopter such as the rotor speed must be correlated with the health indicators to reduce false alarms.*

Keywords: Multi-core, Many-core processor, HMS (Health Monitoring System), Determinism,, Parallelism, Global Algorithms, Incremental Algorithms, Real-time.

Contents

Chapter 1

Introduction

Our goal is to study a real-time avionic function implemented on a many-core processor. This thesis is intended to evaluate a computing platform and a software architecture that provide real-time detection indicators, which could be used by crew.

On-board avionic systems are responsible of diverse functions such as navigation, communication with the ground-station, fuel management, maintenance function etc. They are becoming more and more complex and the airlines companies need to integrate sophisticated functions. Avionic systems are often distinguished by two properties ([1]). First, they are an information processing device. Second, they are *reactive*: they continuously interact with their environment at a speed determined by the environment.

In response to the high-performance integration capabilities, multi-core and many-core processors have first emerged in the *high-performance computing, hardware accelerators community* to improve scalability and/or energy efficiency. However, the main goal in the *embedded systems community* is not to achieve high performance but to verify that the real-time constraints are always met.

The performance gain comes with several challenges when it comes to implementing real-time applications. Determining precise timing guarantees is difficult in many-core processors due to the many contention points. In this thesis, we will evaluate a many-core processor: MPPA-256 from Kalray which has been designed with predictability features: NoC reservation, memory bank configurations.

We choose an avionic function that requires both high processing power and some response time guarantees. The Helicopters *Health Monitoring System (HMS)* performs signal processing on vibration data coming from sensors and raises some mechanical failure alerts to the operating crew. The computation requires a high processing bandwidth and the alerting requires a bounded response time. These characteristics make the HMS a good candidate for an experiment in implementing avionics functions on a many-core processor.

Our purpose is to build an *embedded real-time* implementation of the HMS. The health indicators will be computed on board. Because of the computing requirements of the signal processing algorithms involved, it is necessary to choose an embedded processor that guarantees high performance. In addition, because of the avionics constraints, this processor should also provide low power and predictable response times.

1.1 Thesis Motivations

1.1.1 Different types of maintenance

Improving maintenance is one of the top industrial companies objectives because maintenance issues can lead to safety concerns or systems downtime, and may increase the maintenance costs. There are

different types of maintenance: corrective maintenance, systematic preventive maintenance, conditional preventive maintenance and predictive maintenance. They are defined as follows ([2]):

- Corrective maintenance consists of a set of tasks meant to correct the defects so that the failed machine can be restored to an operational condition. A corrective maintenance is carried out after the detection of the failure.

- Systematic preventive maintenance is carried according to a schedule even if the maintenance is not necessary. There is no intervention before the maintenance date.

- Conditional preventive maintenance is based on the state of the machine. It is performed when some indicators show that the machine is going to fail. The maintenance is only performed when it is necessary.

- Predictive maintenance aims to transform *unplanned maintenance* to *scheduled* corrective maintenance by predicting the future trend of the equipment operations. The prediction is based on statistical approach. The main advantage of the predictive maintenance is to handle: the right information in the right time. It allows to better plan the maintenance activities by knowing when the maintenance is necessary.

The health of the machine is evaluated using non destructive technologies such as infrared, acoustic analysis, vibration analysis, sound level measurements, oil analysis, etc ([2]).

Maintenance improvements in the avionics domain are highly studied in academic as well as in the industry. This thesis is a collaborative work between Airbus Helicopters and the Verimag laboratory. A maintenance model in industry is not necessarily one of the previously mentioned ones. It can be a *mix* of the different maintenance types. The existing HMS is a combination of a systematic and conditional preventive maintenance. We will now describe in more details the HMS.

1.1.2 The Health Monitoring System (HMS) at Airbus Helicopters

The existing HMS function aims to monitor the vibration level measured on the rotating components of the helicopter like the main gear box, the input drive shafts, the tail drive shafts, the intermediate shafts, etc. The function involves two types of sensors: accelerometers and phase sensors. Accelerometers attached to the rotating components measure the vibration signals. Phase sensors allow to synchronize the revolution of the rotor. The vibration signals are acquired during the flight. These signals are sampled and recorded in an acquisition unit. Once the helicopter is on-ground, the data are offloaded and a set of health indicators are computed from the raw vibration data. The computation of health indicators consists of a feature extraction which transform the raw vibration samples into a more informative metrics about the state of the rotating components. The feature extraction is based on signal processing algorithms like Fast Fourier Transform, Hilbert Filter, Mean, Variance, Skewness, etc. A threshold is set, based on human expertise, to determine whether the helicopter needs a maintenance operation or can perform another flight. The maintenance decision is taken after analyzing several hours of flight.

The existing function requires an important network bandwidth to offload data when the helicopter is on-ground. In addition, it reduces the availability of the helicopter because the helicopter is not in service during ground download and analysis. The current function needs an important mass memory to store data during the flight. Finally, the health indicators are computed using all the vibration samples gathered during the flights.

1.1.3 Hardware platforms in Helicopters

IMA standard

A few decades ago, each avionic platform was dedicated to one avionic function. Since the 1990s, the concept of *IMA* ([3]) is used to replace numerous separate platforms by fewer centralized ones. The Integrated Modular Avionics (IMA) concept is a standard system architecture that integrates applications of different criticality on the same hardware platform. The standard enables to integrate several avionic functions sharing platform resources. The shared resources include communication network, I/O. The main goals of the IMA standard is to reduce the weight and the power dissipation. According to [4], Boeing saves 2000 pounds off on the Dreamliner avionic suite thanks to the IMA concept. In the same way, Airbus decreases also by half the number of processors of the avionics suite on the A380.

Hardware/Software Certification Standards

Avionics standards define the rules to be respected in order to certify IMA platforms. The *DO-297* states that applications with different criticality can be executed on the same platform as long as they are isolated. It means applications do not interfere with each other even in case of some failures. Failures that are related to the application, e.g, local bus, deadline, division by 0, should not impact other applications running on the same platform.

Certification authorities have also defined certification rules for avionics software: *DO-178* ([5]). In case of real-time applications, the *DO-178* imposes that the WCET of all applications must be estimated before their executions. The *DO-178* standard provides objectives and gives recommendations to develop avionics software. These objectives relate to verification activities during the software development cycle. During the verification activities, high-level as well as low-level requirements must be checked. As mentioned in [5], a high-level requirement may be: *"The program is never in error state E1"*, and for verifying low-level requirements, such as *"function F computes outputs O1, ..., On from inputs I1, ... Im"*.

Many-core processors for real-time applications

Many-core processors are a potential technology for the aerospace domain. Many-core processors offer the opportunity to investigate new avionic functions that are more time and resource-consuming. In addition to their low power consumption, many-core processors are interesting candidates to replace *IMA (Integrated Modular Avionics)* platforms.

The WCET estimation should not be too pessimistic. Estimating a tight bound worst-case execution time (WCET) for a many-core architecture is very challenging due to the many contention points. These contention points are due to shared resources such as cache, memory, bus, NoC. For example, referring to [6], the table 1.1 shows the memory access latencies for read and write operations while increasing the number of interfering cores. We notice that, on the Freescale P4080, the latency of a read (respectively write) depends on the total number of cores running in parallel (see [7]). The latency of a read operation (write) varies from 41 to 604 cycles (respectively 39 to 1007 cycles).

Shared resources become a bottleneck for high performance and cause predictability issues in execution time. We note that most of the many-core processors are designed taking into account high performance, while the guarantees of a bounded execution are usually not taken into consideration.

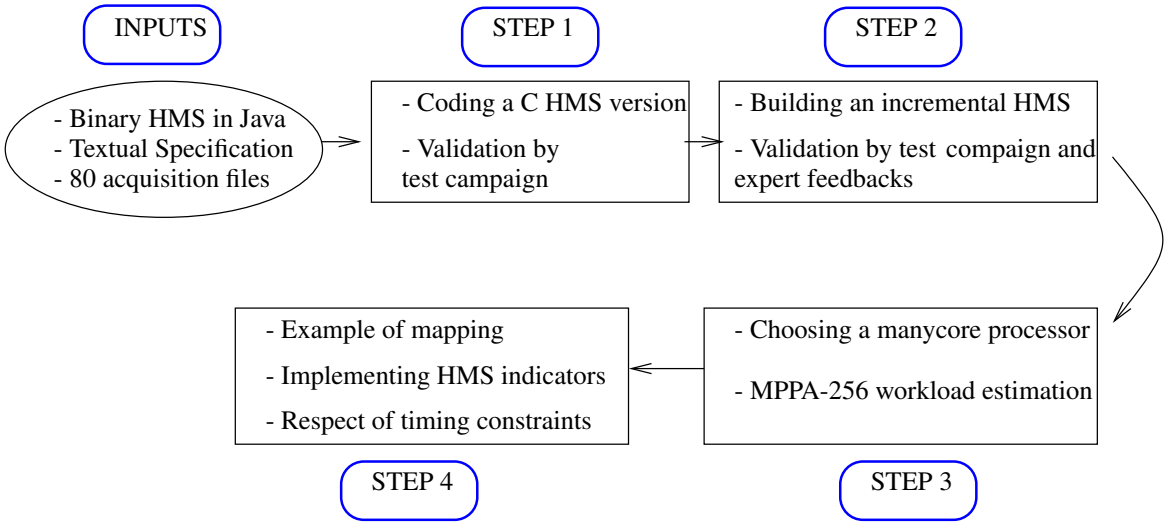
Some modern processors are designed taking into account embedded and real-time constraints. These types of processors reduce contention points from design (they do not remove interferences completely). This is the case of the *Kalray MPPA-256 (Multi-Purpose Processor Array)* that we use in the scope of this thesis.

cores	1	2	3	4	5	6	7	8
read	41	75	171	269	296	439	460	604
write	39	164	245	463	517	737	784	1007

Table 1.1: P4080 memory access latencies in cycles for increasing number of concurrent cores. The Table is extracted from [6]

1.2 Methodology

Figure 1.1 illustrates our approach to build real-time health indicators using the manycore processor MPPA-256.



-

Figure 1.1: Overview of our methodology to build real-time health indicators implemented in C language using a manycore processor from the existing textual specification, 80 raw acquisition file and binary HMS in Java.

As mentioned in Figure 1.1, the inputs are: the existing binary Java version of the HMS, the textual specification of the current HMS and 80 raw files acquired during different flights.

We have first built our C reference version of the existing HMS according to the textual specification. The C reference version is tested with 80 acquisition files. The frequency and temporal health indicators obtained with the two different versions are exactly the same even if we use different library to calculate for instance the interpolation function and Fast Fourier Transform. The first step is detailed in Chapter 3 and Chapter 4.

Chapter 6 covers the step 2 of Figure 1.1. It is our first contribution: the transformation of the C reference version which computes indicators using a full data-set into an incremental on-board computation. Since the MPPA-256 has only 2MB of memory, data are sent by chunk from the acquisition unit to the MPPA-256. The different stages of processing health indicators must respect real-time constraint

given by the volume of vibration samples acquired at the frequency (fs). The health indicators must be processed sufficiently fast before the next acquisition session. The step 2 is validated by comparing the trend of the indicators given by the two different methods and presenting our results to the experts in charge of the maintenance decision.

Step 3 is covered by the Chapters 6, 7 and 8. At step 3, we gave the reasons that have driven our choice for the many-core processor MPPA-256. In addition, we have performed various experiments to evaluate the throughput of one cluster of the MPPA-256. The different measurements aim to evaluate the PCIe bandwidth, the data transmission between the IO cluster and the computing cluster over the NoC, each core is dedicated to one accelerometer. Each core computes in parallel one to three Fast Fourier Transform. Consequently, given 256 accelerometers and a sampling frequency fs= $31kHz$, each cluster is able to compute a workload of 8 FFTs.

Finally, Chapter 9 details the step 4: the implementation of the real-time health indicators on the MPPA-256. Each core no longer computes one Fast Fourier Transform (FFT) but true health indicators. Vibration samples are read from the acquisition unit at each period. The worst case corresponds to the maximum sampling frequency fs=$31kHz$. Each packet of samples is processed in 22 ms which respects the timing constraints equal to 132 ms.

1.3 Contributions

No matter how an advanced health monitoring system is, safety remains a crucial factor in avionics domain. The HMS function should detect in time the mechanical defects. Airbus Helicopters has the ambition to design a HMS function which provides real-time detection capabilities. We aim to answer the following questions:

- How to transform an existing global function into an incremental one?
- How to manage the input flow of vibration samples from the acquisition to the computation of health indicators?
- Which kind of computer is suitable for on-board avionic function?
- How to derive a parallel implementation from a sequential one?

This thesis presents five main contributions to answer these questions:

- **Building an incremental version of the HMS function**: the first contribution covers the transformation of an on-ground reference computation of a health indicators, in which the full data set is used to compute indicators, into an *incremental* embedded real-time computation of the same indicators, in which the computation is updated as soon as data are available.
- **Management of the input flow**: the second contribution focuses on the management of the input flow provided by the sensors to the computation of the indicators. We illustrate how the input vibration data are read by chunks and split to provide the inputs for the computation. We also address the earliest moment when the grouped data are available inside the internal cluster memory and when they are consumed. In addition, we mention the workload estimation of a packet transmitted from the acquisition unit to the MPPA-256. At each step, the maximum amount of data to compute is calculated.

- **Building a dedicated time-stamping tool**: the third contribution tackles a time-stamping mechanism to perform timing measurements on the MPPA-256. The MPPA-256 SDK allows measurements on the simulator which is not completely accurate. The dedicated time-stamping mechanism allows to measure execution duration, latency, bandwidth experiments, verify the predictability mechanisms offered by the many-core processor by addressing the variability of a given measurement repeated many times.
- **Building a software architecture that respects the real-time constraints**: In this contribution, we propose a software architecture that respects the timing constraints. Our measurements consider the maximum frequency and speedup ratio. The mapping covers the Host PC, IO and computing clusters. It is based on a pipeline architecture with three stages in the computing cluster.
- **Illustrating the limits of the HMS function**: the fifth contribution addresses the limits of the current function. We illustrate the limits of the convergence criterion. Our experiments show that the contextual parameters of the helicopter like the speed of the rotor must be correlated to the health indicators in order to reduce false alerts.

1.4 Structure of the document

Chapter 2 exposes the existing health monitoring system: the mechanical system, sensor positions and principles, the acquisition unit. It lists the frequency and temporal indicators and how the maintenance decisions are performed.

Chapter 3 presents the current on-ground HMS global algorithm principle. This chapter discusses the different steps that lead to the computation of indicators.

Chapter 4 details the experiments with the on-ground HMS global algorithm. The experiments are performed with numerous acquisition files and a various flight regime. This step allows to validate our C reference version and to highlight the limits of the existing function.

Chapter 5 exposes the transformation from an on-ground reference computation of health indicators into an incremental version. Different strategies are proposed: sliding or growing windows.

Chapter 6 is a literature review on the existing many-core processors and how to choose a many-core processor for the avionic applications.

Chapter 7 details the advent of many-core processors in the avionics domain. They offer many opportunities to design new avionic functions. However, this gain comes with several challenges.

Chapter 8 presents our preliminary experimental results using a many-core processor. This chapter tackles some timing measurements on the processor, a pipeline software architecture and realistic processing algorithms computation.

Chapter 9 proposes a software architecture to compute real-time health indicators using the many-core processor MPPA-256. The proposed software architecture respects the timing constraint with some measurements on the target and based on the determinism of the processor, we hope that these measurements are very close to the worst case execution time. Finally, it tackles the burst processing due to the variation of the speed of the rotor and a non integer speedup ratio.

Chapter 10 details the different types of maintenance, the health monitoring system in other companies like Sikorsky and AVIC. It also addresses the upcoming technologies in maintenance domain in the industry and finally industrial applications running on the many-core processor MPPA-256.

Chapter 11 presents the perspectives and future works.

Part I

Existing On-ground HMS and Building an Incremental On-ground HMS

Chapter 2

Existing Health Monitoring System (HMS)

The HMS function monitors the vibration of the helicopter system components like gear boxes, transmission shafts, rotors, and bearings. Vibrations are measured by sensors and the data is then provided to a computation unit that performs signal processing to compute health indicators. Some of the HMS indicators are intended to detect mechanical fatigue occurring during helicopter operation.

The algorithms needed to analyze the data provided by vibration sensors are: synchronous average, discrete Fourier transform, reverse discrete Fourier transform, spectrum of welch, Hilbert filter, moment of order x. These algorithms are time- and resource-consuming, especially when they involve the frequency domain.

The current implementation of the HMS is not embedded in the helicopter, and does not need to be computed in real time. An embedded acquisition unit records the vibrations during the flight without any loss (the recording frequency must be at least equal to the sensors sampling frequency). Signal processing computing the various health indicators is performed on-ground, and when the helicopter is on ground. This architecture requires a huge storage capacity (around 256 GB), and a large network bandwidth for data offloading.

The current functional architecture is described in Figure 2.1. Data acquired are stored in a non-volatile memory. The "Processing" box represents the signal processing algorithms involved in the computation of the health indicators.

2.1 Mechanical system, sensor positions

The mechanical system involves phase sensors and accelerometers. Phase sensors are mounted on *reference* shafts. They observe a tooth placed on the rotating part, in such a way that a *top* signal can be generated at each revolution, and sent to the system. Various accelerometers are placed on the rotating elements to monitor their vibrations.

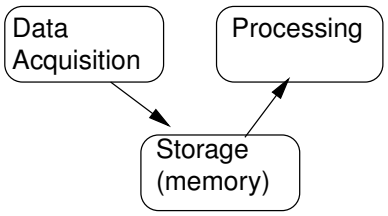


Figure 2.1: Functional Architecture of the HMS

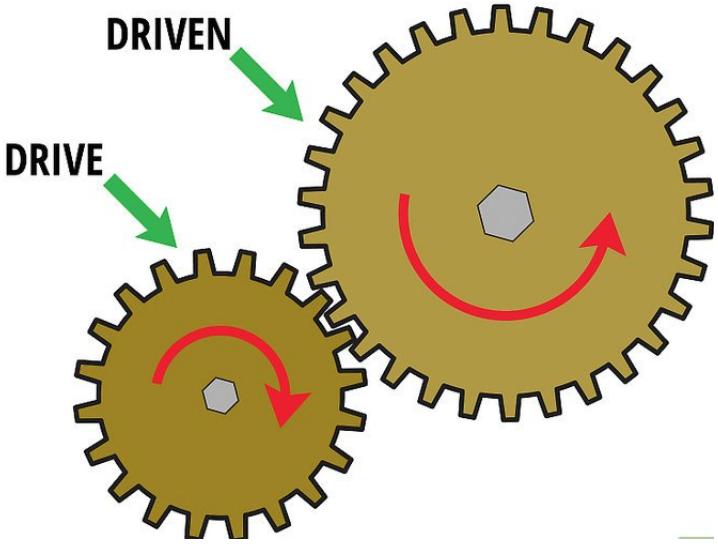


Figure 2.2: The principle of the transmission ratio between two shafts, extracted from [8]

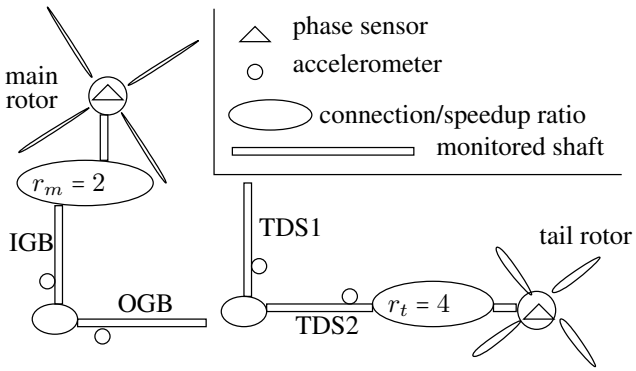


Figure 2.3: The Mechanical System: sensors (accelerometers and phase sensors) and monitored shafts: IGB (Intermediate Gear Box), OGB (Output Gear Box), TDS1 (Tail Drive Shaft 1) and TDS2 (Tail Drive Shaft 2)

Figure 2.2 illustrates the principle of the transmission ratio between 2 wheels: the *drive* wheel and the *driven* one. The term *drive* refers to the wheel connected to the primary shaft from engines. In this example, the drive wheel has z_1 teeth $(z_1 = 20)$ while the driven has z_2 teeth $(z_2 = 30)$. The transmission ratio (r) is defined by the following formula:

$$r = \frac{z_2}{z_1}$$

In this example, $r = \frac{30}{20} = 1.5$. The ratio between the teeth of the two wheels is directly linked to the speed: the drive wheel is 1.5 times faster than the driven wheel.

Figure 2.3 shows two reference shafts: the main and the tail rotor. Two monitored shafts are connected to the main rotor: IGB and OGB, with a speedup ratio $r_m = 2$ (they rotate twice faster than the main rotor). Two monitored shafts are connected to the tail rotor, TDS1 and TDS2, with a speedup ratio $r_t = 4$ (they rotate 4 times faster than the tail rotor).

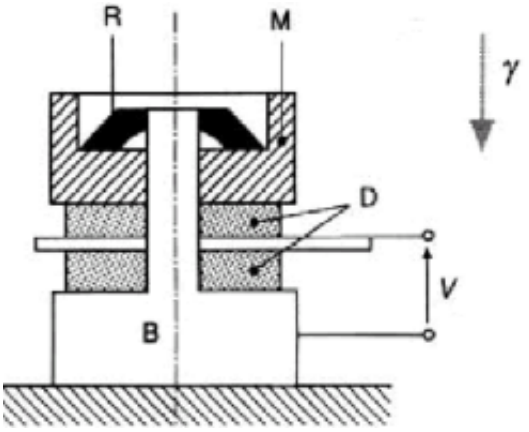


Figure 2.4: The Principle of a piezoelectric accelerometer (extracted from Airbus document), D: piezoelectric element, B: base, V: voltage, γ: acceleration to be measured, R: spring and M: mass

2.2 Sensors principle

The mechanical system of the Health Monitoring System (HMS) is composed of two types of sensors: accelerometers and phase sensors. Accelerometers measure the relative acceleration of the rotating component compared to its initial position. Recall, given the displacement $x(t)$, the acceleration $y(t)$ is calculated as follows:

$$y(t) = \frac{\partial^2 x}{\partial t^2}$$

There exist different types of accelerometers using various technologies. An exhaustive list of the different types of accelerometer technologies is discussed in [9]: piezoelectric accelerometer, optical, magnetic, resonance and so on. Piezoelectric accelerometers are used in the scope of the Health Monitoring System to measure vibrations. A piezoelectric accelerometer transforms a mechanical energy into an electrical one. It generates an electrical current (I) proportional to the acceleration. In general, this current goes through the *signal conditioning step*. Signal conditioning is a way to transform the input signal in a more appropriate wave to be treated thanks to the amplification, filtering and conversion.

- Amplification: it is mainly used when the output signal of the sensor is too low. It consists in a simple multiplication of the input signal with a coefficient K.
- Filtering: it is used to remove some features like noise from the signal.
- Conversion: this step transforms the nature of the signal. For instance, the current can be converted into voltage (V in figure 2.4) thanks to the well known *Ohm law*:

  $$V = R \times I$$

  where R is a resistance and I is the output electrical current proportional to the accelerations.

Figure 2.4 illustrates the principle of a piezoelectric accelerometer.

The principle is the following: when the rotating component has experienced an acceleration, the spring (R) accelerates the mass (M) at the same rate that generates the output voltage (V).

In addition, the phase sensor is a magnetic sensor that delivers a voltage. It has a frequency in the range $[3 - 100]Hz$. This voltage is compared to a threshold to detect rising edges as illustrated in

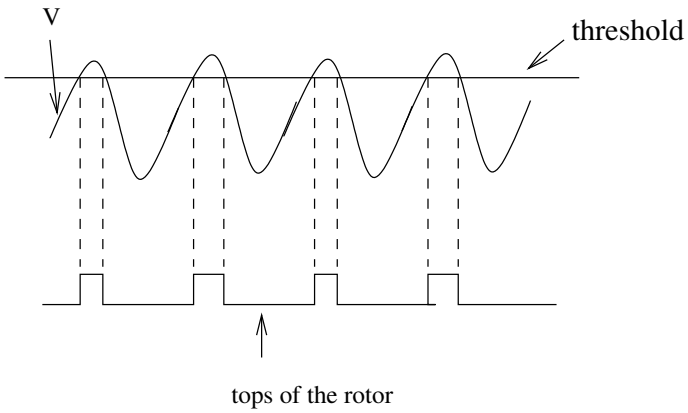


Figure 2.5: From analog output signal of the phase sensor to digital tops

figure 2.5. Each rising edge corresponds to the beginning of a new revolution of the main or tail rotor. In order to detect more precisely the rising edge, the voltage signal is over-sampled at the same rate as the accelerometers between $[1-31]kHz$.

2.3 Acquisition Unit

The acquisition unit is a piece of hardware that reads the sensors, and provides a flow of inputs for the computations. During a flight, the acquisition unit performs automatic acquisition sessions of vibration data. Acquisitions are performed during a stationary flight regime by configuring the sampling frequency and selecting the enabled accelerometers. Each acquisition session is stored in a file and contains vibration samples of accelerometers and tops from the phase sensor. Suppose we have only one phase sensor on the tail rotor, and one accelerometer on a rotating element with speed ratio 4. The acquisition unit builds a sequence of tuples (v, t), where v is a vibration measure, and t is a Boolean, corresponding to the phase sensor. The sequence of values occurring between two successive rising edges of t corresponds to the data gathered during 4 revolutions of the monitored element. The sampling frequency is in the range $[1-31]kHz$. The *top* is very important: although the speed of the rotating parts varies, the *top* allows the measures to be gathered *per revolution* of the monitored rotating part.

Moreover, the data is not delivered one sample at a time. The acquisition unit fills a buffer (the size of which corresponds to 100 KB, typically). The buffer is then read by the computation part, sufficiently often so as not to lose samples.

2.4 Monitored Shaft with different ratios

The HMS monitors several dozens of shafts that rotate at various speeds. Figure 2.6 shows 3 examples of monitored shafts: the *intermediate gear box* illustrated in figure 2.6a, *main gearbox* in figure 2.6b and the *drive shafts* depicted in figure 2.6c.

According to [10], the main gearbox is a mechanical component responsible of transmitting the power from the engines to the rotors of the helicopter. In general, the term *gearbox* (for instance main or intermediate gearbox) refers to a mechanical method that allows to transfer energy from one component to another; it is used to increase torque while reducing the speed. The gearbox is obtained by combining intermediate shafts. Each of these shafts is designed in a specific way in order to generate the required force. According to [11], the speedup ratio between these intermediate shafts is constant and cannot be

Examples of monitored shafts	Speedup ratios	Reference shafts
Input Drive Shaft 1	r_t = 4.005	Tail rotor
Input Drive Shaft 2	r_t = 4.005	Tail rotor
Input Intermediate gearbox	r_t = 4.005	Tail rotor
Output Intermediate gearbox	r_t = 3	Tail rotor
Left MGB input	r_m = 75.26	Main rotor
Alternator Shaft	r_t = 9.58	Tail rotor
Lubrication Pump Shaft	r_t = 2.9	Tail rotor

Table 2.1: Examples of monitored shafts speedup ratios with respect to reference shafts

changed. The gearbox is a complex and critical structure of the helicopter that makes its maintenance very challenging due to the variation of the conditional operation of the helicopter and its complexity. According to [12], the main gearbox causes 10% flight accidents of the helicopter due to its complex structure. In addition, the crack of the main gearbox has killed 45 people in a Boeing helicopter named "slave" in 1986 (refer to [12]). Thus, it is necessary for the flight safety to monitor the health of the different shafts of the helicopter thanks to using sensors and extracting vibrations features from raw data.

Table 2.1 gives examples of monitored shafts with their speedup ratios with respect either to the main rotor (r_m) or to the tail rotor (r_t). Speedup ratios can be *integers,* for example the Output Intermediate gearbox has a speedup ratio of 3 with respect to the tail rotor. They can be also rationals like the lubrication pump shaft which has a speedup ratio of 2.9 with respect to the tail rotor. In addition, there is a disparity of ratios between different shafts. Some run faster than others, sometimes with a huge ratio for instance the *left Main gearbox* rotates 75.26 times faster than the main rotor.

This difference between ratios leads to an organization of the monitored shafts into several categories. Shafts with very close ratios are grouped in the same category. For example, in Table 2.1, Input Drive Shaft 1, Input Drive Shaft 2, and Input Intermediate gearbox could be grouped into the same class. In the same way, Output Intermediate gearbox and Lubrication Pump Shaft could be placed in the same category. Acquisitions are performed by choosing first the category. The idea is to have a sufficiently large number of revolutions of the monitored shafts during the acquisition session. Therefore, each class has its sampling frequency and its acquisition duration. The sampling frequency and the acquisition duration allow to set a *constant* number of samples (*Nbs*) of each sensor (accelerometers and phase sensor) during the acquisition session.

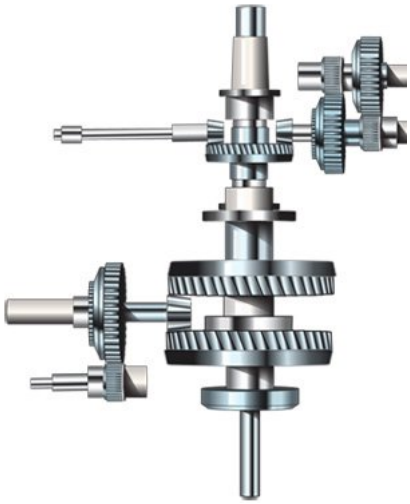
2.5 Different types of indicators: frequency and temporal

Vibration data are used to extract vibration features based on the analysis of indicators. These indicators allow to evaluate the health of the helicopter. Two types of signals are used to calculate indicators: the *raw* signal and the *synchronous average* signal. In the sequel, we denote the raw signal by *rawSignal* and the synchronous average signal by AverageSignal for simplicity. The *rawSignal* is the raw vibration data acquired during the flight. The length of the *rawSignal* depends on the class of monitored shafts previously discussed. It is not directly used to compute indicators. The *rawSignal* is cleaned up when it is acquired in unfavorable conditions for instance during a non-stationary regime of the helicopter. The step corresponding to keeping or rejecting the *rawSignal* is based on the *convergence criteria* discussed in the chapter 3, paragraph 3.6. The convergence criteria calculates a correlation coefficient (CONV) associated to each monitored shaft. This value is compared to a threshold defined for each helicopter. The raw vibration data are *valid* if the correlation coefficient CONV is greater than this threshold.

In addition, the AverageSignal corresponds to a single monitored shaft revolution obtained after

(a) Intermediate gearbox (extracted from Airbus Internal document)

(b) Main gearbox (extracted from Airbus Internal document)



(c) Drive Shafts (extracted from [10])

Figure 2.6: Examples of monitored shafts

averaging of all monitored shaft revolutions in the acquisition file. The different steps that allow to calculate the average signal are detailed in chapter 3, paragraph 3.4.

Once the vibration data are cleaned up, features are extracted by computing indicators. The features extracting step consists in transforming the raw vibration data in a more informative metric directly linked to the state of the machine. The diagnosis of the machine is performed by calculating *temporal* and *frequency* indicators. The appearance of a defect causes significant changes in the statistical and frequency characteristics of the signal.

2.5.1 Temporal indicators

The well known method to perform diagnosis on the rotating machines is to calculate and analyze temporal indicators such as the *Root Mean Square* (RMS), *Kurtosis* (Km), *Skewness* (Sk) and so on.

We suppose that the raw vibration signal (*rawSignal*) of a given monitored shaft is composed of a vector of *Nbs* samples $(v_0 .... v_{Nbs-1})$. The mean value (MEAN_{Nbs}) of the raw vibration signal (*rawSignal*) is calculated as follows:

$$\text{MEAN}_{Nbs} = \frac{1}{Nbs} \times \sum_{i=0}^{Nbs-1} v[i]$$

1. The RMS is the energy average of the vibration signal. It is calculated as follows:

$$\text{RMS} = \sqrt{\frac{\sum_{i=0}^{Nbs-1} (v[i] - \text{MEAN}_{Nbs})^2}{Nbs - 1}} \tag{2.1}$$

   The Root mean square considered here is divided by $Nbs - 1$ instead of Nbs. This value is called the *corrected* root mean square. According to [13], there is a long history in the statistics domain about the denominator $Nbs - 1$ instead of Nbs. Dividing by $Nbs - 1$ allows to get close to the *true* variance of the population.

   The RMS detects cases in which the energy dissipation is abnormally high.

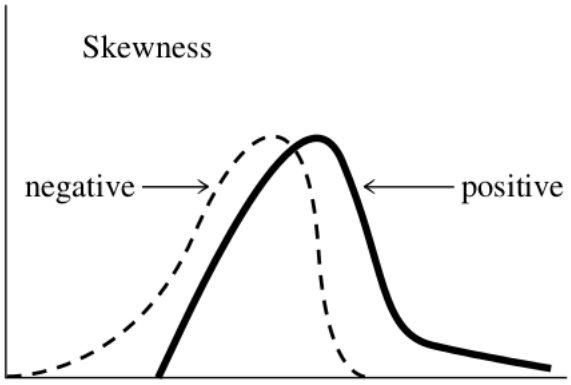


Figure 2.7: Skewness or third moment (extracted from [13])

2. *Skewness* (Sk) or the third moment indicator indicates the degree of asymmetry of the vibration signal around its mean value. Contrary to the mean and the RMS which have the same unit as the measured quantities (here, g unit), the skewness indicator is a *non-dimensional* quantity. It gives an indication about the shape of the distribution of the vibration signal. Figure 2.7 illustrates the interpretation of the skewness value. A negative skewness means that the left tail of the distribution is longer. On the contrary, a positive skewness means that the distribution is concentrated at the beginning and the longer tail is to the right. Sk is calculated as follows:

$$Sk = \frac{1}{Nbs} \times \sum_{i=0}^{Nbs-1} \left[\frac{v[i] - \text{MEAN}_{Nbs}}{\sigma} \right]^3 \tag{2.2}$$

where σ is the standard deviation.

3. *Kurtosis* (Km) or the fourth moment indicator is also a *non-dimensional* quantity. It gives an indication about the flatness of the distribution compared to a normal distribution. According to [13], a distribution with positive kurtosis is named *leptonkurtic* (figure 2.8) while a distribution with negative kurtosis is termed *platykurtic*.

$$Km = \frac{1}{Nbs} \times \sum_{i=0}^{Nbs-1} \left[\frac{v[i] - \text{MEAN}_{Nbs}}{\sigma} \right]^4 \tag{2.3}$$

where σ is the standard deviation.

4. The *RMSR* (Root Mean Square Residual) indicator is calculated as follows:

$$\overbrace{\text{RMS}(\overbrace{\text{FFT}^{-1}(\underbrace{Remove(\underbrace{\text{FFT}(AverageSignal)}_{1}, V)}_{2})}^{3})}^{4} \tag{2.4}$$

$AverageSignal$ denotes the average of all monitored shaft revolutions and V is the set of harmonics to remove from the spectrum. The computation of the RMSR indicator can be divided into four parts. The first part consists in calculating the Fast Fourier Transform using the AverageSignal.

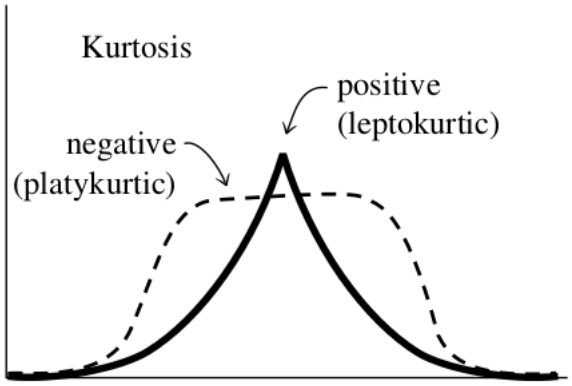


Figure 2.8: Kurtosis or fourth moment (extracted from [13])

During the second step, harmonics specified in the V argument are removed. The third step consists in computing the reverse FFT of the residual frequency spectrum. Finally, the root mean square is calculated using the residual temporal signal.

2.5.2 Frequency indicators

Frequency indicators are becoming a standard in the industry, thanks to the Fast Fourier Transform (FFT) algorithm. The Fast Fourier Transforms ($V(l)$) is calculated as follows:

$$V(l) = \sum_{n=0}^{Nbs-1} v(n) W_{Nbs}^{-nl}, l \in [0..Nbs-1] \tag{2.5}$$

$$W_{Nbs} = e^{j2\pi/Nbs}$$

Applying directly the equation 2.5 requires $O(Nbs^2)$ arithmetical operations of *Nbs* samples .

Frequency indicators refer to indicators calculated in the frequency domain. The Fast Fourier Transform consists in decomposing the energy of a signal into frequency bands. It is very common to associate frequency indicators with temporal ones to analyze the health of the helicopter. It is not rare that some physical phenomena that are very difficult to observe in the temporal domain become more clear in the frequency one. The amplitude and the position of the harmonics constitute a mechanical signature of the state of the machine. Any change about the amplitude or a deviation of the position of the harmonics indicate a probable source of defects. For instance, there is a common way in avionic domain to monitor the operating frequency of rotating components by calculating the OM1 indicator.

1. The OM1 indicator is calculated as follows:

$$\mathrm{OM1} = \overbrace{2 \times \overbrace{\mathrm{SelectHarmonic}(\overbrace{\mathrm{Module}(\underbrace{\mathrm{FFT}(\mathrm{AverageSignal})}_{1})}^{2}, 1)}^{3}}^{4} \tag{2.6}$$

We define that

$$\mathrm{fft} = \mathrm{FFT}(\mathrm{AverageSignal})$$

$$\text{Module(fft,i)} = \sqrt{(Re(\text{fft}[i])^2 + Im(\text{fft}[i])^2)}$$

The indicator OM1 is twice the amplitude of rotation frequency of the monitored shaft. It consists in calculating first the FFT of the AverageSignal signal; then the module of the Fast Fourier Transform of the synchronous average. The third step consists in selecting the 1st harmonic of the spectrum. Finally, the amplitude of the spectrum is multiplied by 2 because the FFT of a real signal is symmetric.

The OM1 indicator allows to detect the imbalance of the rotating shaft. For instance, according to [14], an imbalance issue of the rotor induces difference in the effort of the different blades. The imbalance defect is detected by observing the behavior of the rotating shaft in the frequency domain. It is related to a specific frequency: the rotating frequency of the monitored shaft.

2. The OM2 indicator is calculated as follows:

$$\text{OM2} = \overbrace{2 \times \overbrace{\text{SelectHarmonic}(\underbrace{\text{Module}(\underbrace{\text{FFT}(\text{AverageSignal})}_{1})}_{2}, 2)}^{3}}^{4} \tag{2.7}$$

It consists in calculating first the FFT of the AverageSignal signal; then the module of the Fast Fourier Transform of the synchronous average. The third step consists in selecting the 2nd harmonic of the spectrum. Finally, the amplitude of the spectrum is multiplied by 2 because the FFT of a real signal is symmetric.

OM2 indicator allows to detect the misalignment between two coupling rotating components. A misalignment defect accelerates the deterioration of the machine. According to [15], in industry 30% of time in which the machines are not available is due to poorly aligned machines. Misalignment is estimated to cause over 70% of rotating machinery's vibration problems.

3. The MOD indicator is calculated as follows:

$$\text{MOD} = \underbrace{\text{Modulation}(\underbrace{\text{FFT}(\text{AverageSignal})}_{1}, i)}_{2} \tag{2.8}$$

where the parameter i is an integer number in the range $[1, .., Nbs - 1]$ and

$$\text{Modulation}(\text{fft}, i) = \frac{\text{Module}(\text{fft}[i-1]) + \text{Module}(\text{fft}[i+1])}{2}$$

The MOD indicator is calculated in 2 steps. The first step corresponds to the computation of the FFT of the synchronous average. Then, this step is followed by performing a modulation function around the i^{th} harmonic of the spectrum.

The various shafts are not all monitored with the same indicators.. The choice of temporal or frequency indicators or the combination of the 2 different indicators depend on the monitored shafts. In the same way, the harmonics to consider in the spectrum (for instance, OM1 indicator considers the 1^{st} harmonic) as well as those that must be removed in the case of RMSR indicator are based on the type of

Examples of monitored shafts	Speedup ratios	Reference shafts	Indicators
Input Drive Shaft 1	r_t = 4.005	Tail rotor	OM1,OM2,OM31,MOD31,RMS,RMSR
Input Drive Shaft 2	r_t = 4.005	Tail rotor	OM1,OM2,OM31,MOD31,RMS,RMSR
Input Intermediate gearbox	r_t = 4.005	Tail rotor	OM1,OM2,OM35,MOD35,RMS,RMSR
Output Intermediate gearbox	r_t = 3	Tail rotor	OM1,OM2,OM21,MOD21,OM46,MOD46,RMS,RMSR
Left MGB input	r_m = 75.26	Main rotor	OM1,OM2
Alternator Shaft	r_t = 9.58	Tail rotor	OM1,OM2,OM15,MOD15,RMS,RMSR
Lubrication Pump Shaft	r_t = 2.9	Tail rotor	OM1,OM2,OM40,MOD40,RMS,RMSR

Table 2.2: Examples of monitored shafts speedup ratios with respect to reference shafts and the associated list of indicators

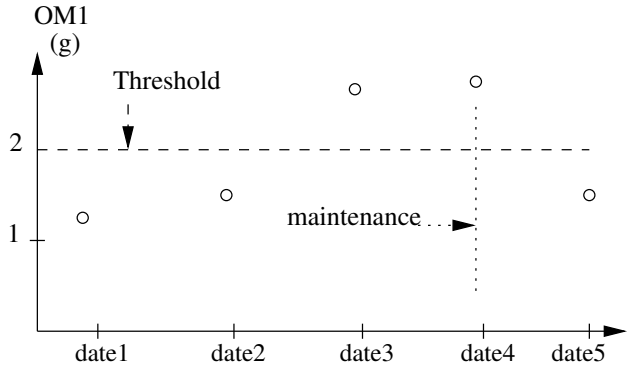


Figure 2.9: Evolution of indicator OM1, for the same flight regime.

shafts and human expertise. Table 2.2 is an extension of Table 2.1 that includes the list of indicators for each monitored shaft. In addition, in Table 2.2, the number of indicators calculated for each monitored shaft is variable. Only two indicators OM1 and OM2 allow to monitor the *Left MGB input* while six are considered to monitor the *Input Drive Shaft 1*.

2.6 Maintenance Decisions

For a given acquisition file and monitored shaft, the existing offline application computes a set of frequency indicators (OM1, OM2, ...) and temporal indicators (RMS, Kurtosis, Skewness, ...). Maintenance decisions are taken by considering the evolution of indicators at successive acquisition dates. Acquisition dates can come from the same flight, or different flights, but always at the same helicopter flight *regime*, in order to be comparable. The helicopter regime is characterized by a value *IAS* (Indicator Air Speed) between *IAS_min* and *IAS_max*, which are configuration parameters for the helicopter.

Figure 2.9 illustrates the evolution of the indicator OM1 depending on the acquisition date. The horizontal axis represents the acquisition dates from 1 to 5, and the vertical axis gives the indicator OM1 (in g unit).

In order to evaluate the evolution of the indicator, a *threshold* value is set, based on experience. If the indicator is above the threshold, it means that a defect has been detected. In figure 2.9, the threshold has been set to $2g$. The value of the indicator has been above $2g$ at dates 3 and 4. A maintenance operation was performed after date 4, and is the reason why the OM1 indicator then decreases to $1.5g$ at date 5.

2.7 Avionic Contextual Parameters

The measured vibration (V_m) during the flight of the helicopter is not only related to the vibrations caused by the mechanical component defects. The measured vibration takes into account the environment (conditional operations), the state of the machines and the pilot operations.

According to [12], the measured vibration has four causes. The first source of vibration is the natural vibration (V_g) caused by the structure of the helicopter itself. The second aspect is related to pilot operations (V_{op}) due to the change of the helicopter fight conditions (Pitch attitude, Roll attitude, Main rotor speed, Free Power Turbine Speed 1, Free Power Turbine Speed 2, Engine Torque 1, Engine Torque 2, Engine Gas Generator Speed 1, Engine Gas Generator Speed 2, Collective stick position, Longitudinal stick position, Lateral stick position, Tail rotor pedal position).

The third source corresponds to the structural damage (V_d). Finally, the fourth source is the vibration coming from other equipment noise (V_n).

Therefore, the measured vibration can be described as:

$$V_m = V_g + V_{op} + V_d + V_n$$

2.8 Conclusion

The current HMS is based on *data analysis* combined with *expertise* by calculating temporal and frequency indicators.

Vibrations change with the flight state and the pilot operations, the evaluation of the helicopter health should take into account contextual parameters. The existing method consisting in setting a fixed threshold value is not very relevant and leads to many false alarms because indicators can go up and exceed the threshold due to the pilot operations. There exist more sophisticated methods in the literature to detect mechanical defects by using maching-learning techniques and considering contextual parameters as illustrated in [12] (this point will be discussed in Chapter 10, page 143). In others words, the existing function does not explore sensor fusions techniques or techniques involving neuronal networks as illustrated in [16]. In addition, the current HMS does not explore a reduction of the volume of data before the computation of the indicators by using for instance a PCA (Principal Component Analysis) as described in [17].

The HMS function provides conditional maintenance of the rotating parts of the helicopter by analyzing the vibration data acquired during several flights. It is a conditional maintenance because the mechanical parts are changed after analyzing the degradation of the machine. A voluntary decision is then made to replace or not the defective components. Today, the conditional maintenance is combined with systematic maintenance. In a systematic maintenance, the defected components are replaced at pre-established time intervals. This maintenance operation requires a history of the indicators on several flights. However, a history of the health indicators is not enough to raise alerts. The diagnosis of failure involves not only the data (the trend of indicators) but also the expertise. There is a trend of combining the human expertise with the artificial intelligence.

In the next chapter, we will describe the different steps that allow to transform the raw vibrations samples acquired during the flight into input samples that will be used to compute indicators. We will show that the extraction of the indicators goes through several steps.

Chapter 3

On-ground HMS Global Algorithm Principle

The existing global algorithm has as inputs binary files containing the vibration data acquired during the flight. Before starting an acquisition session, the number of accelerometers is defined and only one phase sensor is activated (the tail or the main rotor). The acquisition file contains vibration samples of accelerometers and the tops of the phase sensor.

The header of this file also contains general information characterizing the helicopter and the flight for instance the name of the helicopter, the helicopter identifier, the flight identifier, the flight start date and time etc. It also contains information related to the acquisition for example the number of activated accelerometers, their names, sensitivities, the sampling frequency. These general information are only present *once* in the acquisition file, unlike the selected accelerometers and phase sensor that have *Nbs* samples. Recall, *Nbs* depends on the class of the selected sensors during the acquisition session (refer to the end of Paragraph 2.4).

The global algorithm is used to calculate the raw signal that will be used as input to compute the indicators. This step of preparing inputs is done in several stages: detecting the reference tops in the acquisition file, calculating the number of rotations of the monitored shaft, performing an interpolation and a synchronous average.

3.1 Steps of the Global algorithm

3.1.1 Problem statement

There are several shafts to monitor and the indicators which allow to monitor the vibration level are computed using samples of each revolution of a given shaft. Because indicators are associated with monitored shaft revolutions, we must be able to delimit the beginning and the end of each revolution of the monitored shaft. This operation is done by first detecting the reference tops in the acquisition file, then estimating the position of the tops of the monitored shaft, knowing the rotational speed ratio between the reference rotor and the monitored shaft. The implementation of the reference shaft tops detection is shown in Algorithm 3.1 and the implementation of the estimation of the tops of the monitored shaft is illustrated in Algorithm 3.2. These two algorithms allow to gather the samples *per revolution* of the monitored shaft. Notice that, although the sampling frequency is of course constant, since the speed is not constant, the number of samples per revolution varies. A linear interpolation is then used to obtain the same number of values per revolution, for all revolutions of the same acquisition file. A synchronous average provides the raw signal for *one* revolution.

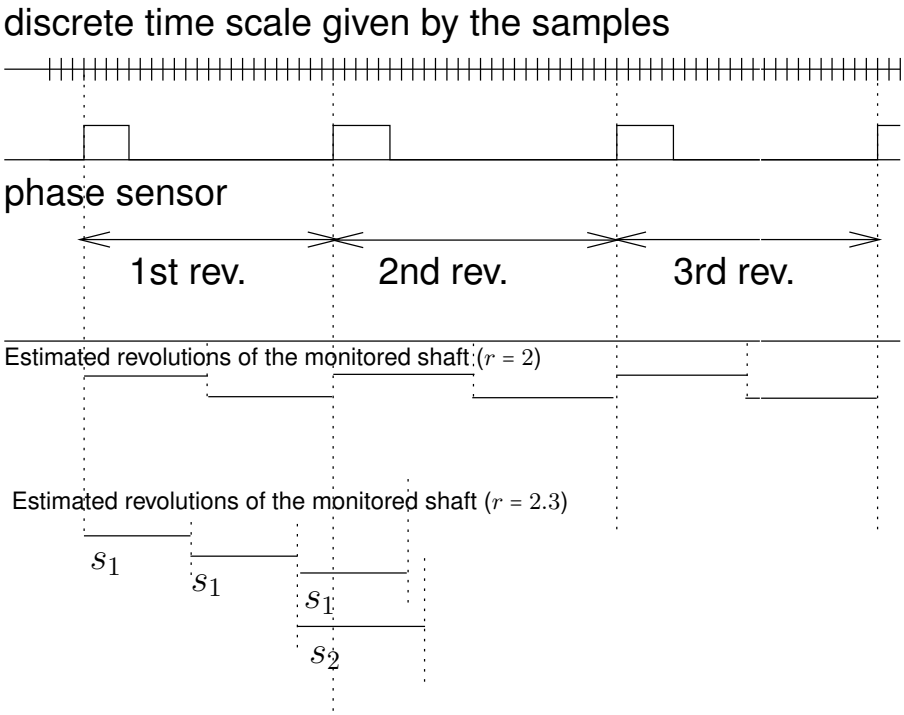


Figure 3.1: Estimated tops of the monitored shaft with integer or non integer speed ratios.

We now examine each of the steps in more details. In the sequel, all the pictures and text are based on the same discrete base scale which is that of the individual samples (the accelerometers and the phase sensor being sampled at the same frequency and synchronously).

3.1.2 Detecting the tops of the reference shaft

The principle of a phase sensor is illustrated on the figure 3.1. A phase sensor signal has two values 0 or 1. Each rising edge indicates the beginning of a new revolution of the reference shaft (main or tail rotor). The rotation speed of the reference shaft is not constant. That is why, in figure 3.1, the interval between two rising edges is not constant. For example, the 2nd revolution of the reference shaft is longer than the 1st revolution. It means the shaft slows down during the 2nd revolution. A lower speed of the reference shaft implies a higher number of samples. Here, in particular, the first revolution has 22 samples, while the second one has 25.

3.1.3 Estimating the tops of the monitored shaft (case of an integer ratio)

Once the positions of the reference revolutions have been detected, each of them has to be split into r equal parts, where r is the speed ratio of the monitored shaft (as shown on Figure 2.3).

On figure 3.1, the first case corresponds to the integer ratio $r = 2$. Each revolution is split into 2 equal parts.

Notice that dividing the reference shaft revolution into *equal* monitored shaft revolutions implies that we consider the speed to be constant during one reference revolution (some knowledge about previous revolutions and a bound on the acceleration could be used to perform a more accurate division, but

the reference global algorithm does not do that). The division consists in placing virtual tops of the monitored shaft on the discrete base scale, as defined above.

3.1.4 Estimating the tops of the monitored shaft (case of an non integer ratio)

Estimating the tops of the monitored shaft is more complicated when the speedup ratio is not an integer. The second part of figure 3.1 shows an example with r = 2.3. The estimated tops of the monitored shaft are no longer aligned with the tops of the reference rotor. As in the previous case we assume the speed is constant during one reference revolution. Hence each of them now has to be divided into 2.3 equal parts. Knowing the number of samples of the 1st revolution, it is divided by r = 2.3, which gives the number of samples s_1 for a monitored shaft revolution.

The problem is that, for the revolution of the monitored shaft that spreads across two successive revolutions of the reference rotor, we cannot assume that the speed is still constant. Hence the first 0.3 portion of the monitored shaft revolution is computed assuming a given speed s_1, and the remaining $(1 - 0.3)$ portion assuming a new speed s_2 (the number of samples of the second revolution, divided by r = 2.3).

Estimating the positions of the virtual tops of the monitored shaft can be done by starting at the beginning of the first reference revolution, and then adding successive segments of the appropriate size. The size of the segment may change at each reference revolution, and a decision has to be made for the size of the segments that spread across two reference revolutions.

In cases like the one just described, the global reference algorithm is based on a quite tricky decision: if the end of the last complete monitored shaft revolution that is entirely included in the reference revolution is *sufficiently close* to the boundary between two reference revolutions, then the first speed (s_1 in the example) is used to determine its length; otherwise the second speed (s_2) is used.

Moreover, in order to determine how close we are to the boundary, the global algorithm is based on the *global average speed*, as observed on the entire acquisition file. This gives the average number of samples that should be observed in a monitored shaft revolution (the size of the segment). If more than half of that number has been observed before the boundary, this means we are *sufficiently close* to the boundary, and we use s_1 to compute the exact length of *that* particular revolution; otherwise we use s_2.

In addition, the number of sample s_1 is first calculated as a *floating* number in order to limit rounding errors. Then, the rounding operations are performed at the end to estimate the positions of the tops of the monitored shaft on the time scale base defined by the samples. Although it's unavoidable to introduce rounding errors in the estimation of the tops of the monitored shaft, we may question the use of the *global average speed* to decide on the length of any particular monitored revolution. The average speed on the two reference revolutions involved could seem more appropriate.

3.2 Algorithm

This algorithm is implemented according to its description in the HMS specification document (see chapter: Existing Health Monitoring System (HMS)). For reasons of clarity, we divide it into two parts: algorithm 3.1, page 36 and algorithm 3.2, page 39. Their purposes are respectively to detect the positions of the reference shaft tops on the time scale defined by the vibration samples and to estimate the positions of the monitored shaft tops.

Notations

- *Nbs*: the accelerometer signal vector length is not constant. The number of samples of each accelerometer for a given acquisition is specified in the HMS specification document. Its value

depends on the category of the selected sensors during the acquisition session (refer to the end of the paragraph 2.4). The array is indexed from 0 to *Nbs* − 1.

- *Ntop*: the number of tops of the reference shaft present in the acquisition file. This counter of tops is initialized at 0 and goes to 1 wen the first top is detected.
- *Nsim*: we decide to consider only the revolutions of the monitored shaft that lie inside complete revolutions of the reference shaft. Consequently, we ignore the first and last (incomplete) revolutions of the reference shaft. Because, the beginning (respectively the end) of the acquisition session does not necessarily coincides with the beginning (respectively the end) of a reference shaft revolution. It is calculated as follows:

$$Nsim = \lfloor (Ntop - 1) \times r \rfloor$$

where $r = r_m$ when the main rotor is the reference shaft and $r = r_t$ when the tail rotor is the reference shaft.

In addition, the speedup ratios are not necessarily integers, we use the *floor* function defined by:

$$\text{floor}(x) = \lfloor x \rfloor$$

to calculate the number of complete revolutions of the monitored shaft. Recall, the floor function gives the largest integer less than or equal to x (refer to [18]).

- REF_MARKER$[i]$: a sample index. It indicates the beginning of revolution $i + 1$ of the reference shaft. The beginning of the first reference shaft revolution in the acquisition file corresponds to REF_MARKER$[0]$.

Example 1. REF_MARKER$[10] = 3405$ means that the 11^{th} revolution of the reference shaft starts at the sample 3405. Note that 3405 is not a vibration value. It corresponds to the *index of the vibration sample at which the magnetic sensor has a rising edge.*

- MONITORED_ROUND_MARKER$[i]$: a sample index that indicates the beginning of revolution $i+1$ of the monitored shaft. The first monitored shaft revolution in the acquisition file corresponds to MONITORED_ROUND_MARKER$[0]$.

Example 2. MONITORED_ROUND_MARKER$[3] = 1100$ means that the 4^{th} revolution of the monitored shaft starts at the 1100^{th} sample on the time scale defined by samples.

- REF_REVOLUTION$[i]$: is an integer. It denotes the number of samples of the reference shaft revolution i (between tops i and $i + 1$). It is calculated as follows:

$$\text{REF\_REVOLUTION}[i] = \text{REF\_MARKER}[i + 1] - \text{REF\_MARKER}[i] \tag{3.1}$$

- MONITORED_REVOLUTION$[i]$: is a *floating* number. . It is calculated as follows:

$$\text{MONITORED\_REVOLUTION}[i] = \frac{\text{REF\_REVOLUTION}[i]}{r} \tag{3.2}$$

The tops of the monitored shaft are first estimated by using floating numbers in order to limit rounding errors. The rounding operations are performed at the end to estimate the index of the beginning and the end of each revolution of the monitored shaft.

- *monitored_revolution_mean*: is a *floating* number which is the mean number of vibration samples of a monitored shaft revolution. This value is computed by considering the full data-set in the acquisition file. It is calculated as follows:

$$monitored\_revolution\_mean = \sum_{i=0}^{Ntop-2} \frac{\text{MONITORED\_REVOLUTION}[i]}{Ntop - 1}$$

  However, this formula is used by the existing HMS and accumulates errors. It can be made more accurate by avoiding intermediate operations, then *monitored_revolution_mean* should be evaluated as follows:

$$monitored\_revolution\_mean = \sum_{i=0}^{Ntop-2} \frac{\text{REF\_REVOLUTION}[i]}{(Ntop - 1) \times r}$$

Assumption

We assume that the speed of the reference shaft is constant during one revolution. Thanks to that assumption, we can apply the previous formula:

$$\text{MONITORED\_REVOLUTION}[i] = \frac{\text{REF\_REVOLUTION}[i]}{r}$$

Algorithm

Recall, the following parameters are associated with a binary acquisition file:

1. speedup ratio (r) of the monitored shaft
2. the selected reference shaft, thus the corresponding phase sensor and its position in the acquisition file (the variable PHASE_BIT in 3.1 indicates the column of samples of the phase sensor).
3. the number of samples acquired during the acquisition session (*Nbs*)

Listing 3.1: Computing the tops of the reference shaft given an acquisition file in a binary format (.hms) using a parser

```
columns
#include "parseAcq.h" // used by the C parser code
#include <stdlib.h>
#include <stdio.h>
#include <math.h>    // for the floor() function

static double monitored_revolution_sum=0; // intermediate variable to calculate "
    monitored_revolution_mean" seen so far
static int nbSample=0; // counter of the number of samples
static int Ntop=0;    // number of tops of the reference shaft seen so far
static int REF_MARKER[NB_SAMPLE]; //sample indices of the reference shaft tops seen so far
static double MONITORED_ROUND_MARKER[NB_SAMPLE]; // real to estimate the sample indices of the
     monitored shafts tops before rounding operations seen so far
static double MONITORED_REVOLUTION[NB_SAMPLE]; // a real to estimate the number of samples of
    each monitored shaft revolution before rounding operations seen sor far
int Nsim; // number of complete revolutions of the monitored shaft seen so far

static void ComputeReferenceTop (UINT8_t id,ANY_t value){
   // id: the pattern we search in the file
   // value: the value of the pattern
   static int previousPhase=1; // value of the previous phase bit
   static int previousTop=0; // position of the previous detected reference top in the time scale
       samples
   static int init=1;       // init allows to consider only complete revolutions of the reference
       shaft
   switch (id){
   case PHASE: // read sensor phase values
      {
      int phase = ((UINT32_t)value)>>(PHASE_BIT)&1; // get the current value of the phase
      if ((phase==1)&&(previousPhase==0)){
         // a rising edge is detected
         if ( init ==0){ // Do nothing for the 1st top
                // Consider only complete revolutions of the reference shaft
            REF_MARKER[Ntop]=nbSample; // set the index of the detected top
            int top = REF_MARKER[Ntop]-previousTop;
            previousTop=REF_MARKER[Ntop];
            MONITORED_REVOLUTION[Ntop-1] = top/r;
            monitored_revolution_sum+=MONITORED_REVOLUTION[Ntop-1];
            Ntop++;  // increment the number of revolutions of the reference shaft
         }
         else {
            // ignore the incomplete revolutions of the reference shaft
             init =0;
            REF_MARKER[0]=nbSample; // get the index of the first top
            previousTop=REF_MARKER[0];
            Ntop++;
         }
      }
      previousPhase = phase;
      nbSample++; // increment the number of samples
      break;
   }
   }

}
```

The main goal of the algorithm 3.1, page 36 is to calculate the index of the tops of the reference shaft in the acquisition file. It provides inputs to the algorithm 3.2 which estimate the tops of the monitored shaft. It is composed of 2 main parts:

- The first part ignores the first incomplete revolutions of the reference shaft. This part goes from line 35 to 41. This part of the algorithm 3.1, page 36 is executed only once when the first reference top in the acquisition file is detected. The first index of the top of the reference shaft is saved and the buffer REF_MARKER[] is initialized (line 38). Notice the samples after the last top of the reference shaft will be ignored.

- The second part corresponds to the core of the algorithm. It goes from line 26 to 34. It is executed for the first time when the second top is detected and every time a new one occurs. This part corresponds to the processing of all complete reference shaft revolutions in the binary acquisition file. REF_MARKER[] is updated every time a new reference top is detected. Thanks to the positions of the tops, the number of samples of each reference shaft revolution is calculated. Then, we estimate the number of samples of each revolution of the monitored shaft (MONITORED_REVOLUTION[]) knowing the speedup ratio (line 31 algorithm 3.1, page 36).

Algorithm 3.2, page 39 has as inputs the results of Algorithm 3.1, page 36: *monitored_revolution_mean*, *Ntop*, REF_MARKER[], MONITORED_REVOLUTION[]. Its goal is to estimate the positions of the tops of the monitored shaft. In order to illustrate the purpose of the algorithm 3.2, let's take a very simple example.

Example 3.

We consider the example illustrated in figure 3.2 where r = 2.3. In addition, we suppose that we have one incomplete revolution and 2 complete revolutions of the reference shaft. The incomplete revolution has 5 samples and the complete ones have respectively 800 and 830 samples per revolution (REF_REVOLUTION[1]=800, REF_REVOLUTION[2]=830). This implies that REF_MARKER[0] = 5, REF_MARKER[1] = 805 and REF_MARKER[2] = 1635. The numbers of samples per revolution of the monitored shaft are respectively s_1 = 347.82 ($\frac{800}{2.3}$) and s_2 = 360.86 ($\frac{830}{2.3}$) (MONITORED_REVOLUTION[0]=347.82 and MONITORED_REVOLUTION[1]=360.86).

The ratio is equal to 2.3. It means that given a reference revolution, the monitored shaft made 2 complete revolutions and an incomplete revolution. This means that the third revolution of the monitored shaft lies across two reference shaft revolutions (refer to figure 3.2). Thus, there will be sets of samples corresponding to one revolution of the monitored shaft, which lie across a top. A question arises: the revolution of the monitored shaft between 2 reference revolutions has 347.82 samples or 360.86 samples?

In fact, if the monitored shaft revolution is totally or nearly covered by the revolution i of the reference shaft, we use REF_REVOLUTION[i] to calculate MONITORED_REVOLUTION[i]. If the monitored shaft revolution is totally or nearly covered by the revolution $i + 1$ of the reference shaft, we use REF_REVOLUTION[$i + 1$] to calculate MONITORED_REVOLUTION[i].

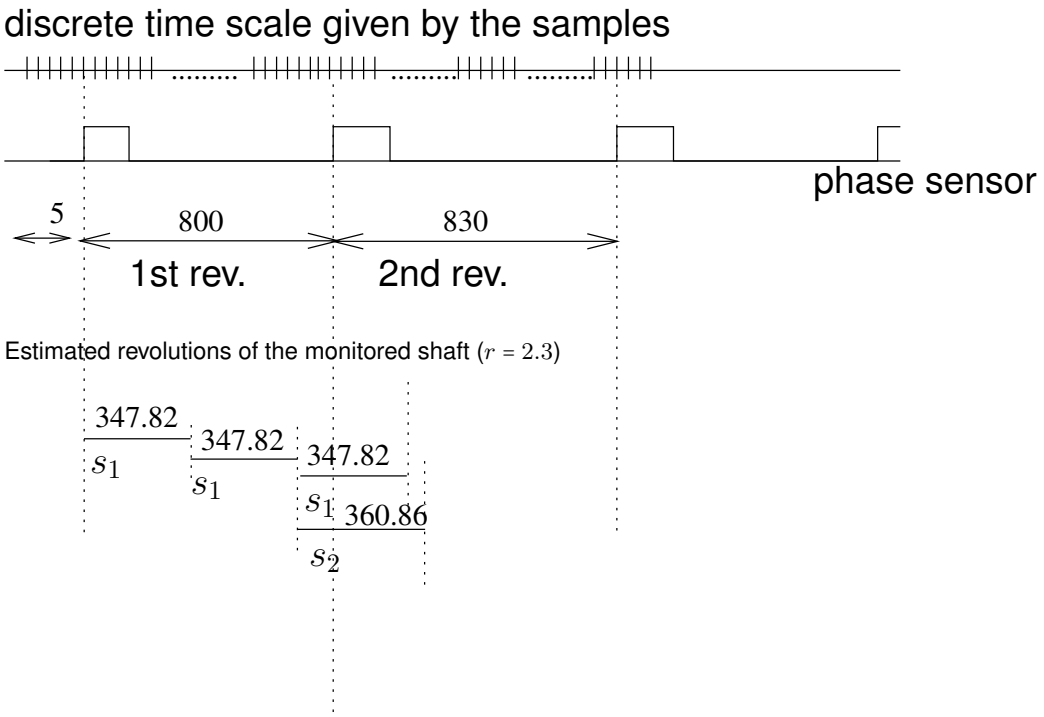


Figure 3.2: Estimated tops of the monitored shaft with a no integer ratio

Listing 3.2: Estimating the tops of the monitored shaft MONITORED_ROUND_MARKER[]

```
1
2  void computeMonitoredTop(double monitored_revolution_sum, int Ntop, int *top_round_marker, double r,
       double *monitored_revolution, double *monitored_round_marker){
3      /* INPUTS:
4      - monitored_revolution_sum: sum of the number of samples of all monitored shaft
           revolutions
5      -Ntop: number of revolutions of the reference shaft
6      - top_round_marker[]: position of the tops of the reference shaft previously
           computed by the function "ComputeReferenceTop"
7      */
8
9      /*OUTPUT:
10     - monitored_round_marker[]: position of the tops of the monitored shaft
11     */
12     double monitored_revolution_mean=monitored_revolution_sum/(Ntop-1); // calculate the mean number
           of samples of all monitored shaft revolutions
13     Nsim = floor(r*(Ntop-1));  //number of revolutions of the monitored shaft
14     int i;
15     monitored_round_marker[0] = top_round_marker[0];
16     int j=0;
17     for (i=0;i<Nsim-1;i++){
18        // estimate the tops of all monitored shaft revolutions
19        // this estimation takes into account the mean number of samples calculated by
              considering the full acquisition file
20        if ((monitored_round_marker[i] < top_round_marker[j]) ||
21                monitored_round_marker[i] > top_round_marker[j+1] - monitored_revolution_mean/2){
22           j = j +1;
23        }
24        monitored_round_marker[i+1] = monitored_round_marker[i] + monitored_revolution[j];
25     }
26 }
```

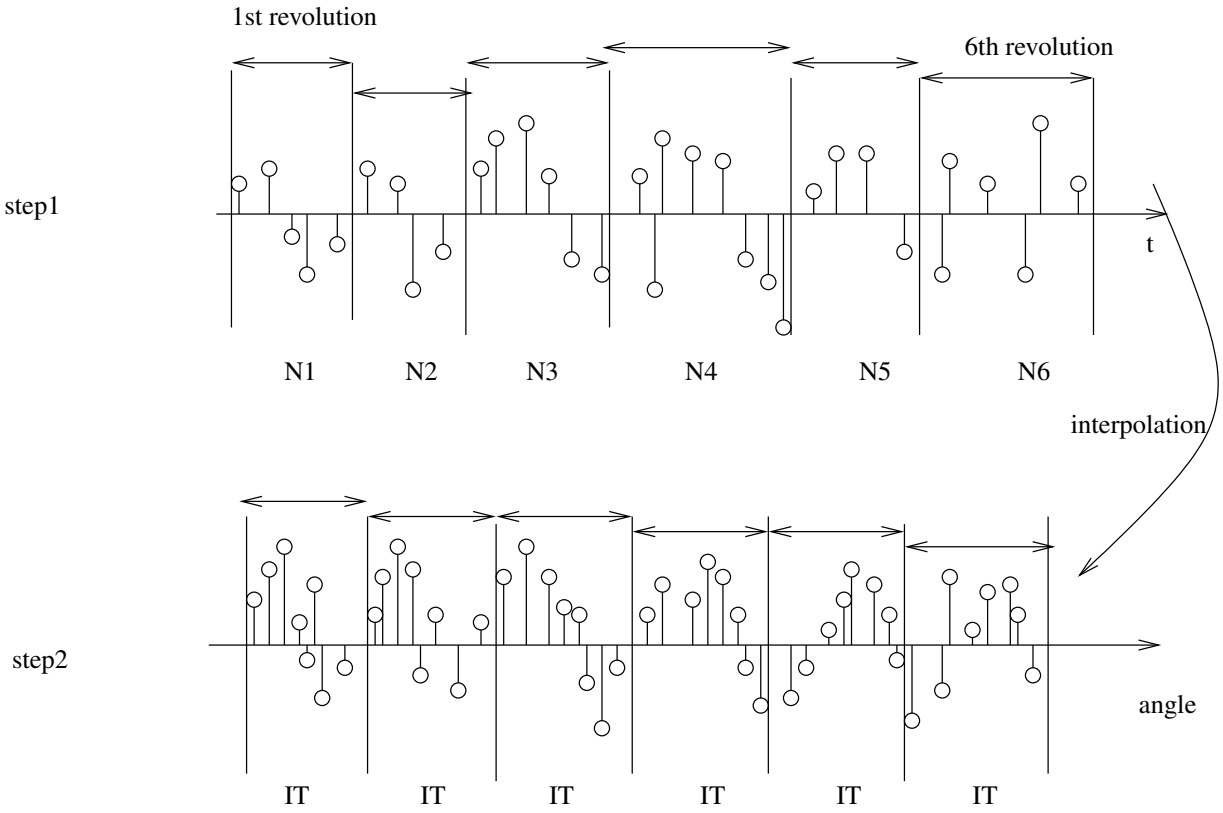


Figure 3.3: Interpolation of each monitored shaft revolution

In the example 3.1 and according to the comparison between MONITORED_ROUND_MARKER$[2]$ and REF_MARKER$[1]$-$\frac{monitored_revolution_mean}{2}$, the number of samples of the third monitored shaft revolution is equal to 360.86.

3.3 Interpolating each monitored shaft revolution

The accelerometer signal depicted in figure 3.3 at step 1 is the same as the signal in figure 3.1 at step 1. We note that the monitored shaft has performed 6 complete revolutions and revolutions have a variable number of samples ($N1, N2, N3, N4, N5$ and $N6$). The interpolation consists in transforming a variable number of samples into a constant number of samples (IT). The on-ground HMS uses a linear interpolation. The number of vibration samples per revolution is given by the interpolation size which is defined in the specification document of the HMS function. We need a constant number of samples for each monitored shaft revolution for many reasons:

1. To associate each sample of the monitored shaft with a constant angle (independently of the variable angular velocity of the shaft) during the revolution of the monitored shaft in order to be able to average the value of this sample over several revolutions.

2. Computing indicators with a constant number of samples allows to optimize execution time.

   Suppose that we need to evaluate the spectral power density of the frequency fe with a sampling frequency (fs) and a number of samples N, the index k of the frequency fe in the DFT (Discrete Fourier Transform) is given by:

$$k = \frac{fe}{fs * N} \tag{3.3}$$

Because the rotation velocity w of the shaft is variable, fe and N are also variable and so is k. This leads to an efficiency issue (recalculation of k and coefficients of the DFT).

On the contrary, we consider a finite sequence with N constant samples

$$x(n), n \in [0..N-1]$$

We define its discrete Fourier transform as the sequence $X(l)$:

$$X(l) = \sum_{n=0}^{N-1} x(n) W_N^{-nl}, l \in [0..N-1] \tag{3.4}$$

$$W_N = e^{j2\pi/N}$$

We can avoid recalculating the twiddle factors W_N^{-nl} every time that new data $x(n)$ arrives. To do that, it is more efficient to pre-compute once the twiddles factors W_N^{-nl} depending on N.

3. For accurate amplitude detections. In fact, the amplitude of the vibration signal depends on the rotational frequency of the rotor. Harmonics of vibrations are compared to a constant threshold. If the angular velocity varies, the amplitudes of vibration harmonics also vary. It is impossible to set an amplitude threshold independent of the monitored frequency.

4. To decrease spectral leakage:

   The vibration signal ($\mathrm{x}(t)$) is continuous. The acquisition session is equivalent to observing the vibration signal during a limited duration. In signal processing, this operation is called: windowing a function. According to [19], a windowing function ($\mathrm{w}(t)$) is a mathematical function that is zero-values outside of some chosen interval. The simplest window function is a rectangular window. It is constant inside an interval and zero elsewhere. The windowing function gives us the signal $\mathrm{x}_w(t)$:

   $$\mathrm{x}_w(t) = \mathrm{x}(t) \times \mathrm{w}(t)$$

   and the corresponding Fourier Transform is given by the multiplication/convolution theorem:

   $$\mathrm{X}_w(f) = \mathrm{X}(f) * \mathrm{W}(f)$$

   Example 1. Consider the example described in [20].

   $$\mathrm{x}(t) = sin(4 \times \pi \times t) + 0.2 \times sin(8 \times \pi \times t)$$

   $\mathrm{x}(t)$ is composed of two sinusoids of different frequencies ($2Hz$ and $4Hz$). The theoretical Fourier transform of $\mathrm{x}(t)$ is given by the green wave in figure 3.4. It is composed of 4 peaks at $4Hz$, $-4Hz$, $2Hz$ and $-2Hz$. However applying a rectangular window involves some difference in the spectral waveform (blue wave). In addition to the 4 peaks, the window function produces a non-zero value between peaks (spectral leakage). In this example, we note that, because of the spectral leakage,

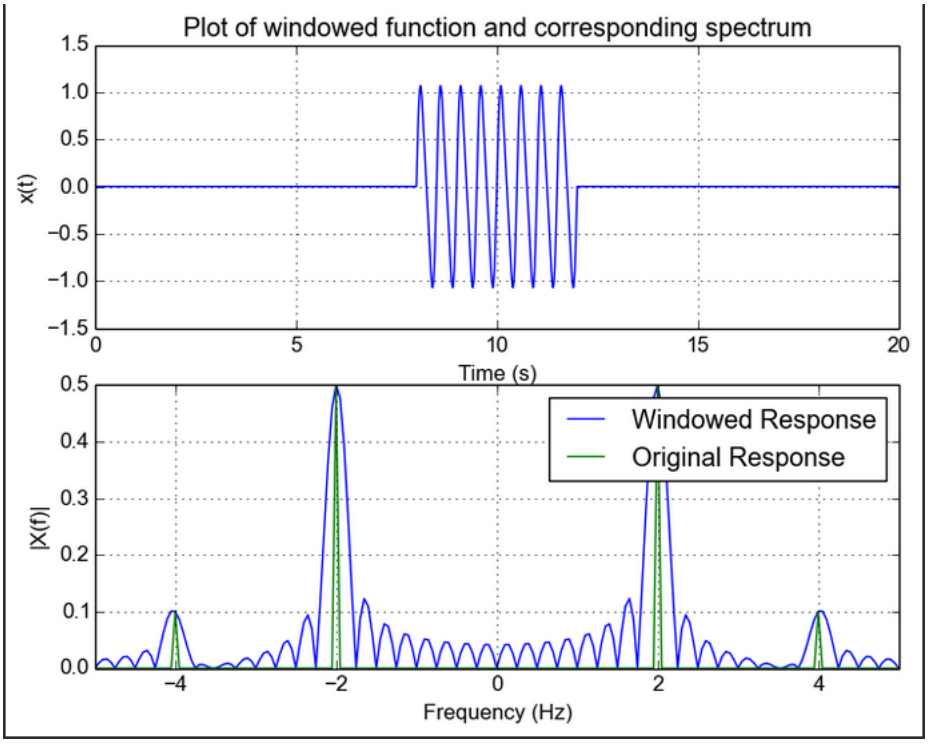


Figure 3.4: Spectral leakage

it becomes difficult to distinguish the peak at $4Hz$ and $-4Hz$. The leakage coming from the strong peaks at $2Hz$ and $-2Hz$ obscures the weak peaks ($4Hz$, $-4Hz$).

According to [20], in the case of a rectangular function, increasing the number of samples reduces the effects of leakage. Furthermore, as a general case, increasing the number of samples is the best way to decrease spectral leakage regardless of the kind of window function. Except in some rare cases, the interpolated signal size is greater than the number of samples per revolution of the monitored shaft. Thus, the interpolation increases the number of samples per revolution of the monitored shaft.

3.4 Average signal

Once the tops of the monitored shaft are calculated, each monitored shaft revolution is interpolated as illustrated in figure 3.3. At this stage, all the monitored shaft revolutions have the same number of samples per revolution (IT). We define *Nsim* as the total number of monitored shaft revolutions in the acquisition file. All the revolutions of the monitored shaft are averaged and this gives as result a single revolution representing the average signal as described in figure 3.5. In other words, this phase allows to move from *Nsim* revolutions to a single revolution and therefore to reduce the noise.

3.5 Computing Indicators

Finally, indicators are computed using the average signal of the monitored shaft revolutions. In other words, the acquisition file contains several monitored shafts revolutions but the indicators are computed

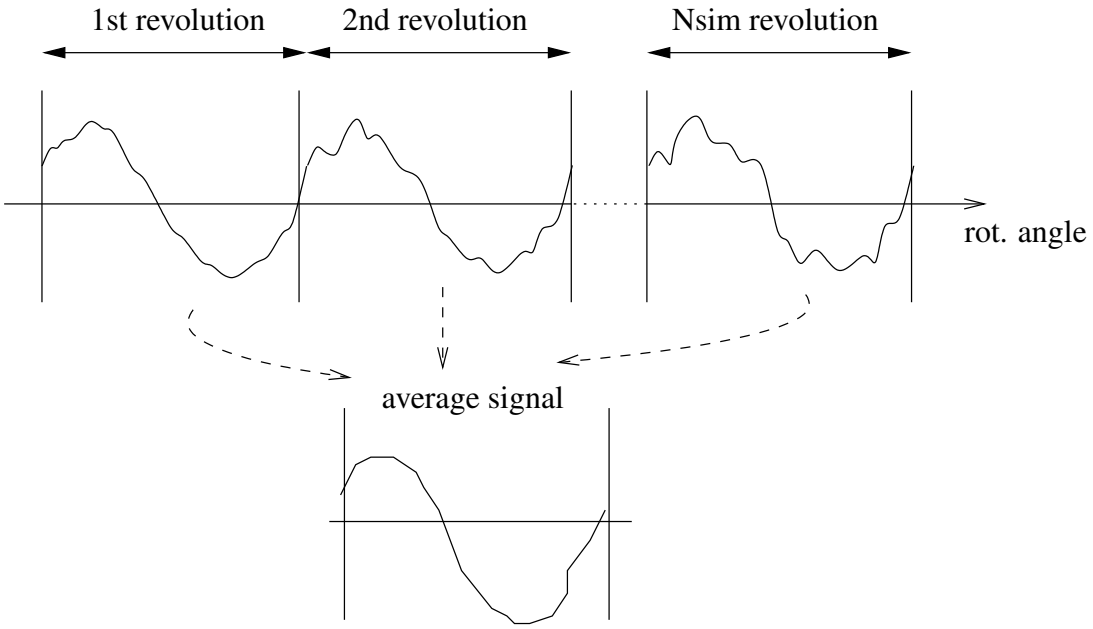


Figure 3.5: Synchronous averaging of monitored shaft revolutions

for *one* monitored shaft revolution which is the average of all revolutions in the acquisitions file. It means, given an acquisition file and a monitored shaft, only one value of each indicator is computed even if the acquisition file contains *Nsim* revolutions (refer to Paragraph 2.6).

3.6 Convergence Criteria

Principle

The convergence criterion allows to validate an acquisition session. In addition to performing acquisitions during particular flight phases, a second filter is used to decide whether the acquired data will be exploited or not. This convergence criterion is calculated over several monitored shaft revolutions. This criterion represents the covariance between the average of the first $Nsim/2$ revolutions and the average of all *Nsim* revolutions. The result of the covariance is compared to a threshold $limit$. A covariance equals to -1 means that the two signals are in opposition phase whereas a covariance equals to 1 means that the two signals are identical, $limit$ is set around 0.8 in the HMS specification document.

Algorithm

The inputs of the convergence formula are as follows:

- X: the average synchronous signal for *Nsim* revolutions of the monitored shaft. X is computed according to figure 3.5.
- Y: the average synchronous signal for $\frac{Nsim}{2}$ revolutions of the monitored shaft. Y is computed according to figure 3.5.
- IT: the interpolation size
- X_m: the mean value of X

- Y_m: the mean value of Y
- S_x: the standard deviation of X
- S_y: the standard deviation of Y

The output of the convergence rate CONV is given by:

$$\text{CONV} = \frac{1}{\text{IT}} \times \sum_{i=1}^{\text{IT}} \frac{(X_i - X_m)(Y_i - Y_m)}{S_x \times S_y} \tag{3.5}$$

3.7 Conclusion

This chapter has discussed the different steps that lead to the computation of indicators. These indicators are calculated by using either the raw vibration signal (*rawSignal*) or the average signal (AverageSignal). The extraction of the average signal (AverageSignal) from the raw vibration samples has been detailed in this chapter. It has four main steps: detecting the tops of the reference shaft, estimating the tops of the monitored shaft, the interpolation step and the average signal.

The existing HMS version is implemented in *Java* and we have only the binary. We design our own reference version in *C* language by coding the textual specification of the *Java* one. It is important to notice that, when estimating the beginning and end of a monitored shaft revolution, we compute the position as an integer index on this time scale. This involves some rounding operations. The experiments described in Paragraph 4.9, page 61, have shown that the overall computation of the health indicators is not too sensitive to these rounding operations, with the existing on-ground application.

As mentioned in Paragraph 2.6, page 29, the vibration samples are not acquired in any conditions but in a stationary flight regime. Acquiring data in specific conditions among all the flight domain of the helicopter is like filtering data. In addition, the convergence criteria is used as a second filter.

In the following chapter, we will calculate examples of temporal indicators like (RMSR) and frequency ones for instance (OM1, OM2). These indicators are calculated with several acquisition files that monitor different shafts. These files are also acquired during various flight conditions. Furthermore, we will validate our *C* implementation by comparing the indicators results with the *Java* ones given the same input acquisition files. Finally, we have shown that the 2 filters (acquiring data in specific conditions and the convergence criteria) are insufficient to compare indicators with data acquired in the same conditions. The contextual parameters should be considered.

Chapter 4

Experiments on the On-ground version of the HMS Global Algorithm

The same specification (refer to the pseudo algorithms in Section 3.1, page 36 and in Section 3.2, page 39) for the existing application which is in *Java* (we have only the Java binary) is used to develop a new version in *C* language. This step allows to have our own reference version with which the real-time indicators calculated on board will be compared. It allows also to prepare the code that will be embedded on the many-core MPPA-256 processor.

In order to validate the new reference version in C, we compared the 2 versions (Java binary and C) with several acquisition files from sensors that monitor different shafts. They are acquired in different flight regimes. The number of test files is 80. We cannot create test files because the validation test must be performed with true vibration samples. If differences are observed, they could come from data types (float, double) and libraries for example to compute math functions like *linear interpolation*.

4.1 Experiments Overview

The main parameters of each acquisition file are:

- the number of activated accelerometers. Each of these accelerometers can monitor one or several mechanical shafts. These shafts rotate at different angular speeds (refer to 2.2).
- the selected phase sensors, thus the reference shafts (main or tail rotor).
- the sampling frequency (fs).
- the number of vibration samples (*Nbs*) of each sensor during the acquisition session.

Before comparing the final results of the indicators calculated in two different ways (Java and C), we will first focus on the intermediate stages (detecting the reference tops in the acquisition file, calculating the number of rotations of the monitored shaft, performing an interpolation and a synchronous average).

The specification document of the current HMS states that: acquisitions are performed during a particular flight phase: *Steady Flight Level (SFL)*. According to [21], the SFL regime named also *equilibrium* flight is defined as a flight stage in which there is no acceleration. In others words, the angular speed of the rotor is constant. We will first perform a series of experiments in order to verify the flight regime of the helicopter during the acquisition sessions. This verification is done by evaluating the variation of the speed of the two reference shafts during the acquisition phase. For this, we need to divide all the vibration samples of a given shaft into reference shaft revolutions. Dividing vibration samples into reference revolutions is made by detecting rising edges of the top of phase. The number of samples of each reference shaft revolution is calculated by counting the number of samples between two rising

edges. Thus, by knowing the number of samples of each revolution of the reference shaft and the sampling frequency, it is possible to deduce the angular speed associated with each revolution. By angular speed, we refer to the number of Revolutions Per Minute (RPM) of the reference shaft. We will consider 3 acquisition files acquired during different flight regimes.

- **case 1**: very slight acceleration followed by a very slight deceleration of the reference shaft using vibration data of the *Input Drive Shaft 1*.
- **case 2**: quasi stationary angular speed of the reference shaft using the vibration data coming from the *Tail Drive Shaft 1*.
- **case 3**: regular acceleration of the reference shaft using the vibration data of the *Tail Drive Shaft 2*.

These experiments have shown that the rotor speed is *not strictly constant* during the acquisition phase. There are cases in which the speed variation of the rotor exceeds *8%*. The rotor speed goes from 1149 to 1252 RPM. In fact, data acquired in such conditions should not be considered in order to compare data acquired in the same conditions (SFL regime). As a reminder, the convergence criterion is used to filter data acquired during regular acceleration or deceleration phase. However, vibration data in case 3 satisfies the existing convergence criterion. Consequently, these experiments have also shown the limit of the existing convergence criterion.

Then, because the health indicators are associated with monitored shaft revolutions, we experiment the split of each reference revolution into monitored ones using the speedup ratio. We compare the start and end indices of each monitored shaft revolution obtained with the Java and C versions. The 2 versions give the same indices (at the discrete scale given by the samples) for all revolutions of the monitored shaft. We have repeated this test several times on different monitored shafts and with acquisition files acquired at different flight conditions such as case 1, case 2 and case 3. These tests allow us to validate our C implementation of the intermediate stage: estimating the positions of the tops of the monitored shaft.

Once the vibration samples are divided into monitored shaft revolutions, we focus on the interpolation step of each monitored shaft revolution. The existing version uses an Apache Java library to compute a linear interpolation. However, our C reference version implements an interpolation function without using any external library. We have performed unit tests of these two implementations. These tests are performed with different interpolation sizes (256, 512 and 1024). Given the same input samples, the linear interpolation calculated in two different manners is the same.

In addition, we have performed a series of experiments to calculate indicators. Our choices are driven by two key points. First, we need to cover all the types of indicators: frequency such as OM1, OM2 and temporal ones like RMSR. Second, the indicators are computed by using as input either the synchronous average signal (OM1, OM2, RMSR) or the raw vibration data like RMSb indicator. Each of these indicators is calculated using 20 raw acquisition files. These acquisition files contain the vibration samples of the *input drive shaft 1* which has a ratio equal 16.82 with respect to the tail rotor. The temporal and frequency indicators calculated by the 2 implementations are the same.

Finally, we have noticed that estimating the monitored shaft indices introduces rounding errors because the ratios are real numbers and the rotor speed during two consecutive revolutions varies. We have evaluated the sensitivity of the indicators by shifting the indices of the beginning and the end of each monitored shaft revolution by a few positions. This operation has not a great impact on the indicators.

The following sections detail each of these experiments. We will discuss about the purpose of each experiment and its input parameters. We will also tackle how to implement it and finally its result.

4.2 Dividing an acquisition file into reference shaft revolutions

4.2.1 Structure of an acquisition file

Figure 4.1 shows an example of an acquisition file. This file contains vibration samples of 2 accelerometers and tops of one phase sensor. It is structured on 3 columns (the first two columns are vibration samples and the last one is coming from the phase sensor). At each sampling period ($\frac{1}{\text{fs}}$), one sample of each sensor is produced. For instance val_0_1 corresponds to the second vibration sample of sensor 0.

4.2.2 Test parameters of the reference top detections

The algorithm of the reference top detections already discussed in Section 3.1 is tested with real acquisition files acquired during different flights of the helicopter $H175$ from Airbus Helicopters. These tests are initially performed with a standard PC (not on the MPPA-256). Table 4.2 illustrates the results of the tops detection algorithm using vibration data of the *input drive shaft 1* which has the *tail rotor* as a reference shaft. The acquisition file contains 32 reference tops (*Ntop* = 32). This means that the tail rotor has completed 31 complete revolutions during the acquisition session. Furthermore, the acquisition file we consider in this experiment contains *Nbs* = 50000 samples for each sensor and the sampling frequency fs = 31250 *Hz*.

4.2.3 Experimental Results of the reference top detections

REF_MARKER$[2]$ = 4385 in Table 4.2 denotes that the beginning of the 3^{rd} revolution of the tail rotor starts at sample 4385. More generally, REF_MARKER$[i]$ indicates the beginning of revolution $i + 1$ of the reference shaft.

The first reference top is detected at the 1283^{th} sample and the last one at 49317^{th} position. This means that the first 1282 samples and the last 683 $(50000 - 49317)$ samples in the acquisition file belong to incomplete revolutions of the tail rotor.

4.3 Experiments characterizing the helicopter flight regimes

4.3.1 Principle of characterizing the flight regime of the helicopter

The current HMS function performs acquisition sessions during a particular flying stage: stationary flight regime. This stage happens when the speed of the rotor is almost constant. We will first focus on the recognition of the flight regime by calculating the speed of the rotor during the acquisition session. Two parameters are needed to evaluate the speed of the rotor: the sampling frequency and the number of samples of each revolution of the rotor.

The sampling frequency (fs) is already known because it is associated with the acquisition session. Thus, we will focus on how to count the number of samples of each revolution of the reference shaft.

In fact, the phase sensor allows to delimit the revolutions of the reference shaft. The samples gathered between two rising edges correspond to one revolution of the rotor (main or tail). The right-hand part of Figure 4.1 depicts the signal produced by the phase sensor. Each rising edge indicates the beginning of a new revolution of the reference shaft.

The on-ground HMS algorithm iterates through all the samples of the chosen accelerometer and indicates the positions of the phase sensor. The vibration samples between 2 rising edges belong to the same reference shaft revolution.

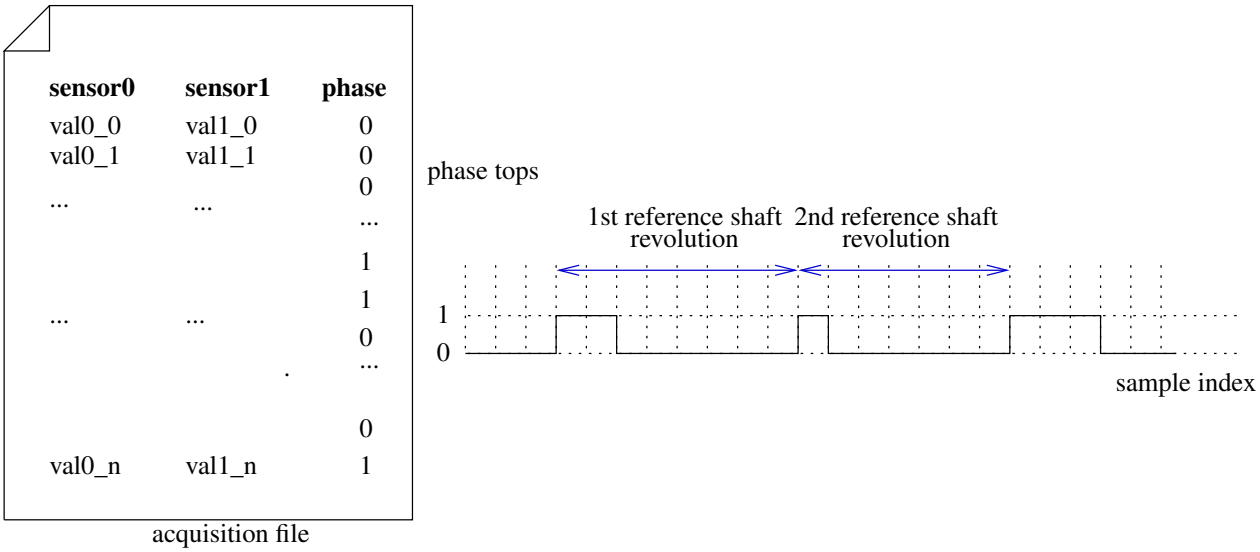


Figure 4.1: Acquisition file with 2 accelerometers and one phase sensor and the principle of reference tops detection

REF_MARKER$[i]$	sample indices	REF_MARKER$[i]$	sample indices
REF_MARKER$[0]$	1283	REF_MARKER$[1]$	2834
REF_MARKER$[2]$	4385	REF_MARKER$[3]$	5936
REF_MARKER$[4]$	7487	REF_MARKER$[5]$	9037
REF_MARKER$[6]$	10587	REF_MARKER$[7]$	12137
REF_MARKER$[8]$	13686	REF_MARKER$[9]$	15236
REF_MARKER$[10]$	16786	REF_MARKER$[11]$	18336
REF_MARKER$[12]$	19887	REF_MARKER$[13]$	21436
REF_MARKER$[14]$	22986	REF_MARKER$[15]$	24534
REF_MARKER$[16]$	26082	REF_MARKER$[17]$	27631
REF_MARKER$[18]$	29179	REF_MARKER$[19]$	30729
REF_MARKER$[20]$	32277	REF_MARKER$[21]$	33826
REF_MARKER$[22]$	35374	REF_MARKER$[23]$	36921
REF_MARKER$[24]$	38470	REF_MARKER$[25]$	40019
REF_MARKER$[26]$	41569	REF_MARKER$[27]$	43118
REF_MARKER$[28]$	44668	REF_MARKER$[29]$	46218
REF_MARKER$[30]$	47767	REF_MARKER$[31]$	49317

Figure 4.2: Reference top indices calculated using an acquisition file acquired during H175 flight

The algorithm for detecting the tops of the reference shaft is implemented in C according to its specification (see Algorithm 3.1). It takes as input an acquisition file and produces an array (REF_MARKER[]) that contains the tops indices of the reference shaft in the acquisition file.

The number of samples for each revolution i is then computed:

$$\text{REF\_REVOLUTION}[i] = \text{REF\_MARKER}[i+1] - \text{REF\_MARKER}[i] \tag{4.1}$$

And finally, the rotor speed in RPM is calculated as follows:

$$\text{SPEED\_RPM}[i] = 60 \times \frac{\text{fs}}{\text{REF\_REVOLUTION}[i]} \tag{4.2}$$

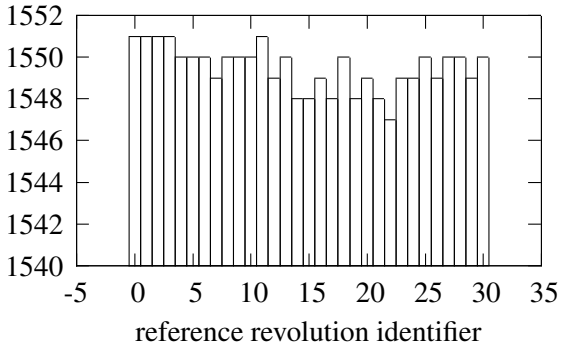


Figure 4.3: The number of samples of each revolution of the tail rotor using vibration data of the *input drive shaft 1*

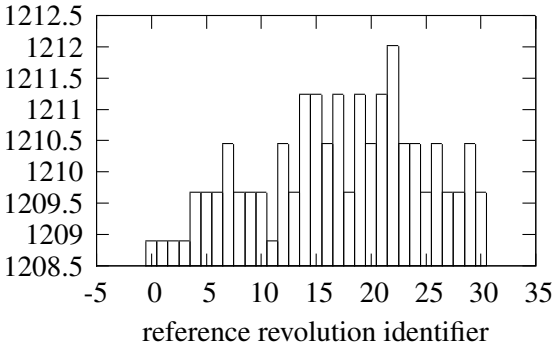


Figure 4.4: Evolution of the speed of the tail rotor (in RPM) during the acquisition session

4.3.2 Experiments on characterizing the flight regimes

We repeat the previous experiment on several acquisition files. The following experiments address three differences that we consider to be representative because they highlight the different characteristic of the acquired data. First, data may come from various sources (sensors). Then, the reference shaft can be either the main or the tail rotor. Finally, the data can be acquired in various conditional operations like when the acquisition is performed during an acceleration phase.

Case 1: Very slight acceleration followed by a very slight deceleration

This experiment is performed using the vibration data of the monitored shaft *input drive shaft 1*. This data is acquired during very slight acceleration followed by a very slight deceleration of the rotor.

Test Parameters. The monitor shaft is the *input drive shaft 1* with $r_t = 16.82$. The acquisition file contains 32 reference tops (*Ntop* = 32). This means that the tail rotor has completed 31 complete revolutions during the acquisition session. This file contains *Nbs* = 50000 samples for each sensor and the vibration data is acquired at fs = 31250*Hz*.

Results. We plot the number of samples per revolution of the tail rotor in figure 4.3. We observe that the number of samples per revolution is not constant. The number of samples per revolution is between

1547 and 1551. This relative gap is equal to 0.25%. In addition, figure 4.4 depicts the evolution of the speed of the tail rotor (number of Revolutions Per Minute: RPM) for each revolution. The speed is calculated according to equation 4.2. The speed varies between 1208.5 and 1212.5 RPM. Thus, the speed of the tail rotor is not strictly constant. During the acquisition session, the tail rotor accelerates and decelerates very slightly.

Case 2: Quasi stationary angular speed of the rotor

This experiment is performed using the vibration data of the monitored shaft *tail drive shaft 1*. This data is acquired when the speed of the rotor is quasi-stationary.

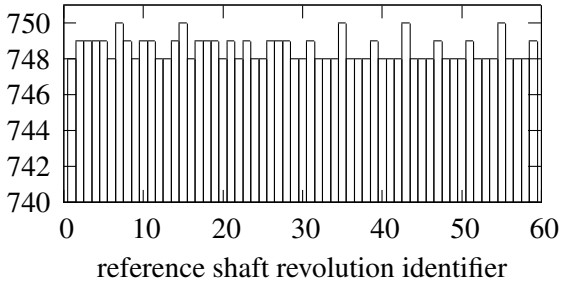


Figure 4.5: The number of samples of each revolution of the tail rotor using vibration data of the *tail drive shaft 1*

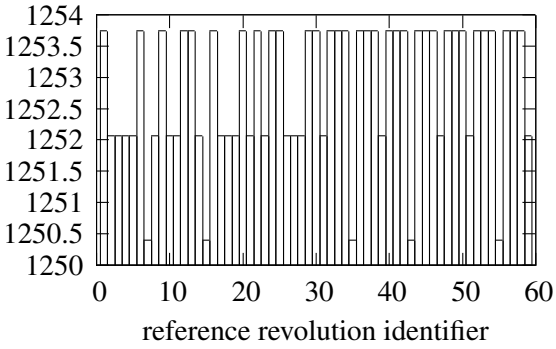


Figure 4.6: Evolution of the speed of the tail rotor (in RPM) during the acquisition session of the tail rotor using vibration data of the *tail drive shaft 1*

Test Parameters. In contrast to the previous experiment, the following one depicted in figure 4.5 concerns the *tail drive shaft 1* which has the *tail rotor* as reference shaft. Vibrations are sampled at fs = 15630Hz and the number of samples recorded is equal to *Nbs* = 45000.

Results. We note that in figure 4.5, the reference shaft has performed 59 complete revolutions during the acquisition session. The number of samples is between 748 and 750. The number of samples per revolution of the reference shaft varies slightly. In the same way, figure 4.6 depicts the evolution of the speed of the rotor during the acquisition session. The rotation speed is quasi-stationary. Thus the rotation speed variation is equal to 0.266% during the acquisition session.

The two previous experiments count the number of samples per reference shaft revolution, the number of revolutions of the reference shaft during the acquisition phase and finally we observed that the rotational speed was almost stationary during the acquisition phase. We will show in the following experiment that the rotational speed variations of the reference shaft can be much greater than those previously observed.

case 3: Regular acceleration

This experiment is performed using the vibration data of the monitored shaft *tail drive shaft 2*. Vibration data is acquired during a regular acceleration regime.

Test Parameters. Experiment in figure 4.7 allows to monitor the *tail drive shaft 2* which has the *tail rotor* as reference shaft. Vibrations are sampled at fs = $15630Hz$ and the number of samples recorded is equal to *Nbs* = 45000.

Results. Concerning the figure 4.7, the number of samples per reference revolution varies slightly but much more than the two previous experiments. The number of samples is between 749 and 816. The number of samples per revolution decreases progressively during the acquisition session. Furthermore, the rotational speed illustrated in figure 4.8 shows that the rotational speed increases and goes from 1149 to 1252 RPM. In others words, the variation of the speed of the rotor is equal to 8.2% during the acquisition session.

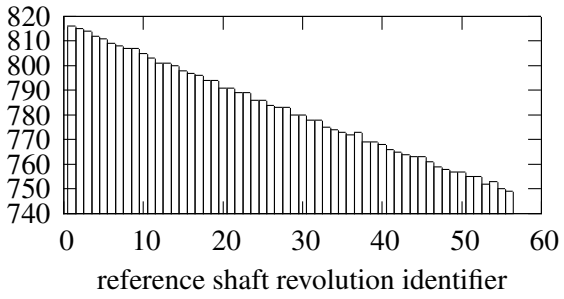


Figure 4.7: The number of samples of each revolution of the tail rotor using vibration data of the *input drive shaft 2*

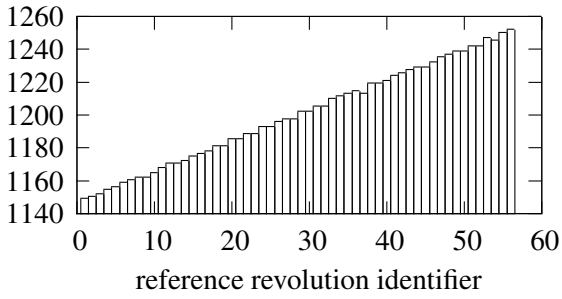


Figure 4.8: Evolution of the speed of the tail rotor (in RPM) during the acquisition session using vibration data of the *input drive shaft 2*

4.3.3 Conclusion

Previous experiments used acquisition files from different sensors. But only the phase sensor is used. The phase sensors mounted on the main and the tail rotors allow to highlight several points including the variation of the number of samples of the reference shafts; thus the rotational speed of the reference shafts are not constant. The convergence criterion described in Paragraph 3.6 is used to clean up data acquired during regular acceleration such as case 3 or deceleration.

In the following experiments, the vibration data coming from the sensors are combined with the phase tops to illustrate the different steps that derive to the computation of the indicators.

4.4 Dividing a reference shaft revolution into monitored shaft revolutions

The purpose of this experiment is to estimate the positions of the tops of the *input drive shaft 1*, i.e. to estimate the positions of the samples which correspond to the beginning (MONITORED_ROUND_MARKER$[i]$) and the end (MONITORED_ROUND_MARKER$[i+1]$) of revolution i of the *input drive shaft 1*. The positions of the tops of the *input drive shaft 1* are estimated using Algorithm 3.2.

MONITORED_ROUND_MARKER$[i]$ – MONITORED_ROUND_MARKER$[i+1]$	indice boundaries
MONITORED_ROUND_MARKER$[0]$ – MONITORED_ROUND_MARKER$[1]$	[1283.000000-1375.203280]
MONITORED_ROUND_MARKER$[1]$ – MONITORED_ROUND_MARKER$[2]$	[1375.203280-1467.406561]
MONITORED_ROUND_MARKER$[2]$ – MONITORED_ROUND_MARKER$[3]$	[1467.406561-1559.609841]
MONITORED_ROUND_MARKER$[3]$ – MONITORED_ROUND_MARKER$[4]$	[1559.609841-1651.813122]
MONITORED_ROUND_MARKER$[4]$ – MONITORED_ROUND_MARKER$[5]$	[1651.813122-1744.016402]
MONITORED_ROUND_MARKER$[5]$ – MONITORED_ROUND_MARKER$[6]$	[1744.016402-1836.219682]
MONITORED_ROUND_MARKER$[6]$ – MONITORED_ROUND_MARKER$[7]$	[1836.219682-1928.422963]
MONITORED_ROUND_MARKER$[7]$ – MONITORED_ROUND_MARKER$[8]$	[1928.422963-2020.626243]
MONITORED_ROUND_MARKER$[8]$ – MONITORED_ROUND_MARKER$[9]$	[2020.626243-2112.829523]
MONITORED_ROUND_MARKER$[9]$ – MONITORED_ROUND_MARKER$[10]$	[2112.829523-2205.032804]
MONITORED_ROUND_MARKER$[10]$ – MONITORED_ROUND_MARKER$[11]$	[2205.032804-2297.236084]
MONITORED_ROUND_MARKER$[11]$ – MONITORED_ROUND_MARKER$[12]$	[2297.236084-2389.439365]
MONITORED_ROUND_MARKER$[12]$ – MONITORED_ROUND_MARKER$[13]$	[2389.439365-2481.642645]
MONITORED_ROUND_MARKER$[13]$ – MONITORED_ROUND_MARKER$[14]$	[2481.642645-2573.845925]
MONITORED_ROUND_MARKER$[14]$ – MONITORED_ROUND_MARKER$[15]$	[2573.845925-2666.049206]
MONITORED_ROUND_MARKER$[15]$ – MONITORED_ROUND_MARKER$[16]$	[2666.049206-2758.252486]
MONITORED_ROUND_MARKER$[16]$ – MONITORED_ROUND_MARKER$[17]$	[**2758.252486-2850.455766**]

Table 4.1: Estimating the positions of the tops of the input drive shaft 1

4.4.1 Test Parameters

In this experiment, we consider the same acquisition file as the reference tops detection (table 4.2). The monitored shaft is the *input drive shaft 1* and its rational speed with respect to the tail rotor is $r_t = 16.82$.

4.4.2 Results

Table 4.1 depicts the positions of the first 17 revolutions of the input drive shaft 1. We note that the first revolution of the monitored and reference shaft begin at the same sample 1283. As a reminder, the first reference revolution starts at position 1283 and ends at 2834 as described in Table 4.2 (REF_MARKER$[0]$ = 1283 and REF_MARKER$[1]$ = 2834). This reference revolution must be split into 16.82 parts which gives the number of samples s_1 for a monitored shaft revolution. The number of samples s_1 is calculated as follows:

$$s_1 = \text{MONITORED\_ROUND\_MARKER}[0] = \frac{\text{REF\_REVOLUTION}[0]}{r_t} = 92.20328$$

where REF_REVOLUTION$[0]$=2834-1283=1551 and $r_t = 16.82$. As the ratio is $r_t = 16.82$, the first 16 monitored shaft tops in the first reference revolution are calculated as follows: the first monitored top is calculated (MONITORED_ROUND_MARKER$[0]$=1283), the segment s_1 is added to MONITORED_ROUND_MARKER$[0]$ to estimate the next monitored shaft top (MONITORED_ROUND_MARKER$[1]$=1375.203280), s_1 is also added to MONITORED_ROUND_MARKER$[1]$ to estimate the next virtual top (MONITORED_ROUND_MARKER$[2]$) and so on.

The 17^{th} monitored shaft revolution which spreads across two reference revolutions is estimated using Algorithm 3.2, page 39, which basically consists in using either s_1 as the previous virtual tops or s_2 to calculate MONITORED_ROUND_MARKER$[17]$. In fact, in the table 4.1, s_1 is used to estimate MONITORED_ROUND_MARKER$[17]$ because the 16^{th} virtual top (MONITORED_ROUND_MARKER$[16]$=2758.252486) is *sufficiently* close to the boundary (MONITORED_ROUND_MARKER$[1]$=2834).

$round(\text{MONITORED\_ROUND\_MARKER}[i]) - round(\text{MONITORED\_ROUND\_MARKER}[i+1])$	rounded indices
$round(\text{MONITORED\_ROUND\_MARKER}[0]) - round(\text{MONITORED\_ROUND\_MARKER}[1])$	[1283-1375]
$round(\text{MONITORED\_ROUND\_MARKER}[1]) - round(\text{MONITORED\_ROUND\_MARKER}[2])$	[1375-1467]
$round(\text{MONITORED\_ROUND\_MARKER}[2]) - round(\text{MONITORED\_ROUND\_MARKER}[3])$	[1467-1560]
$round(\text{MONITORED\_ROUND\_MARKER}[3]) - round(\text{MONITORED\_ROUND\_MARKER}[4])$	[1560-1652]
$round(\text{MONITORED\_ROUND\_MARKER}[4]) - round(\text{MONITORED\_ROUND\_MARKER}[5])$	[1652-1744]
$round(\text{MONITORED\_ROUND\_MARKER}[5]) - round(\text{MONITORED\_ROUND\_MARKER}[6])$	[1744-1836]
$round(\text{MONITORED\_ROUND\_MARKER}[6]) - round(\text{MONITORED\_ROUND\_MARKER}[7])$	[1836-1928]
$round(\text{MONITORED\_ROUND\_MARKER}[7]) - round(\text{MONITORED\_ROUND\_MARKER}[8])$	[1928-2021]
$round(\text{MONITORED\_ROUND\_MARKER}[8]) - round(\text{MONITORED\_ROUND\_MARKER}[9])$	[2021-2113]
$round(\text{MONITORED\_ROUND\_MARKER}[9]) - round(\text{MONITORED\_ROUND\_MARKER}[10])$	[2113-2205]
$round(\text{MONITORED\_ROUND\_MARKER}[10]) - round(\text{MONITORED\_ROUND\_MARKER}[11])$	[2205-2297]
$round(\text{MONITORED\_ROUND\_MARKER}[11]) - round(\text{MONITORED\_ROUND\_MARKER}[12])$	[2297-2389]
$round(\text{MONITORED\_ROUND\_MARKER}[12]) - round(\text{MONITORED\_ROUND\_MARKER}[13])$	[2389-2482]
$round(\text{MONITORED\_ROUND\_MARKER}[13]) - round(\text{MONITORED\_ROUND\_MARKER}[14])$	[2482-2574]
$round(\text{MONITORED\_ROUND\_MARKER}[14]) - round(\text{MONITORED\_ROUND\_MARKER}[15])$	[2574-2666]
$round(\text{MONITORED\_ROUND\_MARKER}[15]) - round(\text{MONITORED\_ROUND\_MARKER}[16])$	[2666-2758]
$round(\text{MONITORED\_ROUND\_MARKER}[16]) - round(\text{MONITORED\_ROUND\_MARKER}[17])$	[**2758-2850**]

Table 4.2: Rounding the estimated positions of the tops of the input drive shaft 1

4.5 Gathering samples per monitored shaft revolution

The vibration data of accelerometers in the acquisition file are binary values. These values are obtained after sampling the voltage signal proportional to the acceleration (refer to Paragraph 2.2). The digitalization phase is composed of two steps: sampling phase and the quantification. After sampling the continuous signal, its sample has a numerical value according to its amplitude (quantification). In order to cover all the possible values of the amplitude, this number is coded in 16 bits. Then, the following formula gives the vibration sample in mV:

$$\text{V(mV)} = \frac{\text{V(binary)} \times 2 \times \text{InputRange}}{2^{16} - 1} \tag{4.3}$$

where V(binary) is the vibration value in binary stored in the acquisition file and InputRange is the range of the output voltage set before performing an acquisition.

Indeed, the acceleration unit in the international system is m/s^{-2}. It corresponds to the acceleration caused by the force of gravitation. We use the *g* unit instead of m/s^{-2} in order to differentiate the gravitational acceleration and the acceleration caused by the speed variation of an object. Expressing acceleration in *g* unit allows to directly compare the acceleration of a moving object to the gravity. Thus, according to [22], 1g is the acceleration of gravity on the surface of the earth. For instance, 2g means the moving object undergoes the equivalent of 2 times the normal gravity on earth.

In the case of an accelerometer, the vibration in g unit can be deduced from the vibration in mV using the equation 4.3 and the sensitivity of the accelerometer (mV/g) by the following formula:

$$\text{V(g)} = \frac{\text{V(mV)}}{\text{sensitivity(mV/g)}} \tag{4.4}$$

Given an acquisition file, the existing global algorithm selects first an accelerometer and stores its vibration data in an array named *VibrationSamples*. *VibrationSamples* contains all the vibration data of that accelerometer during the acquisition session. Its size is equal to *Nbs*.

Listing A.1 details the different steps to gather all the vibration samples of a given accelerometer in the array *VibrationSamples*. First, we must select which accelerometer to consider by defining the

value of the variable SENSOR_ID. The range of SENSOR_ID depends on the number of activated accelerometers (NUMBER_ACCEL) during the acquisition session. Then, we need two additional values (accelerometer sensitivity and input range) in the acquisition file to calculate each vibration sample in g unit according to equation 4.4. Finally, line 38 of Listing A.1 gathers all the samples of the selected accelerometer.

In the previous experiment and thanks to Algorithm 3.2, we have estimated the indices of the monitored shaft *input drive shaft 1* as illustrated in Table 4.1. Once, the positions of the tops are estimated, we can gather samples per monitored shaft revolution. Becausse arrays are accessed using integer index, the real indices (MONITORED_ROUND_MARKER$[i]$) must be rounded to have integer indices.

In addition, the positions of the tops of the monitored shaft are rounded by using the round function: *ROUND()*. According to [23], the round function returns the nearest integer value of the argument passed to this function. Table 4.2 shows the results when applying the round() function to the real indices estimated in the previous experiment.

4.6 Interpolation of the raw vibration signal of the monitored shaft revolution

In the previous sections, we have shown how to split an array of samples of a given accelerometer into monitored shaft revolutions. In addition, we have also addressed the need of using integer index to access every single revolution of the monitored shaft. Then, recall, every single array of samples of a given monitored shaft is interpolated in a constant number of samples (Section 3.3).

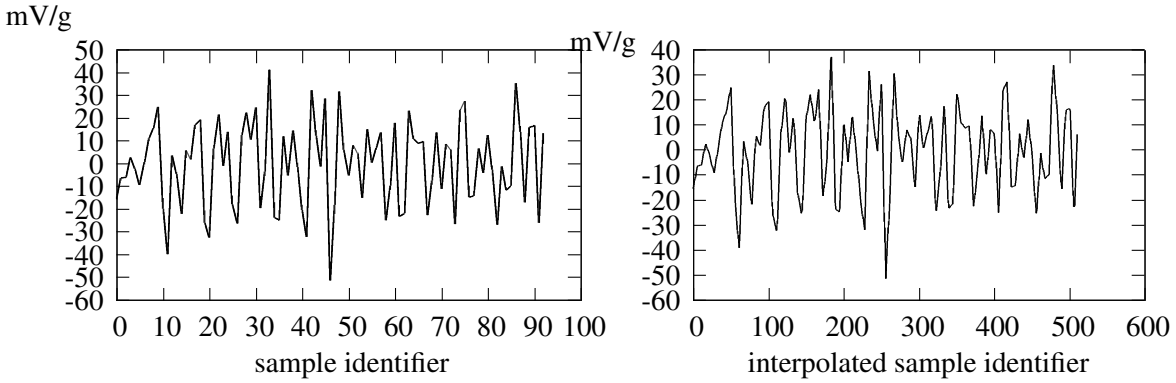
The current interpolation function is implemented in *Java* by using the math *Apache* library ([24]). Its prototype is given in Listing A.2. It uses a polynomial spline function to calculate the interpolation. However, the existing HMS uses the spline function as a linear function.

We have implemented a linear interpolation function in our *C* reference version without using any library. This implementation is shown in Listing A.3. This algorithm takes as input an array of samples corresponding to one revolution of the monitored shaft and builds an interpolated array of IT samples. A linear function is an easy way of getting values at positions between the input data points and its requires only two points (refer to line 20 and 21 of Listing A.3). These points are joined by straight lines. According to [25], the linear interpolation induces discontinuities at each point. A *cosine* interpolation produces smoother transitions between linear segments and therefore requires more than 2 points. However, our first goal is not to propose more relevant interpolation function but to reproduce the existing one. We have compared the outputs of the two different implementations of the linear interpolation. The output interpolated samples are always the same when repeating the test many times with different acquisition files. These tests allow us to validate our *C* linear interpolation implementation.

Figures 4.9 and 4.10 illustrate the interpolation of the first two revolutions of the monitored shaft. Figures 4.9a and 4.9b represent respectively the raw vibration signature of the first monitored shaft and its linear interpolation.

The first monitored shaft revolution is depicted in figure 4.9a and has 92 samples. The measured vibration level is between -50 and 40mV/g. Then this revolution is interpolated on 512 samples. In this case, the linear interpolation transforms a raw signal of 92 samples to another signal built on 512 samples (figure 4.9b). The interpolated signal has the same level of vibration and the same shape.

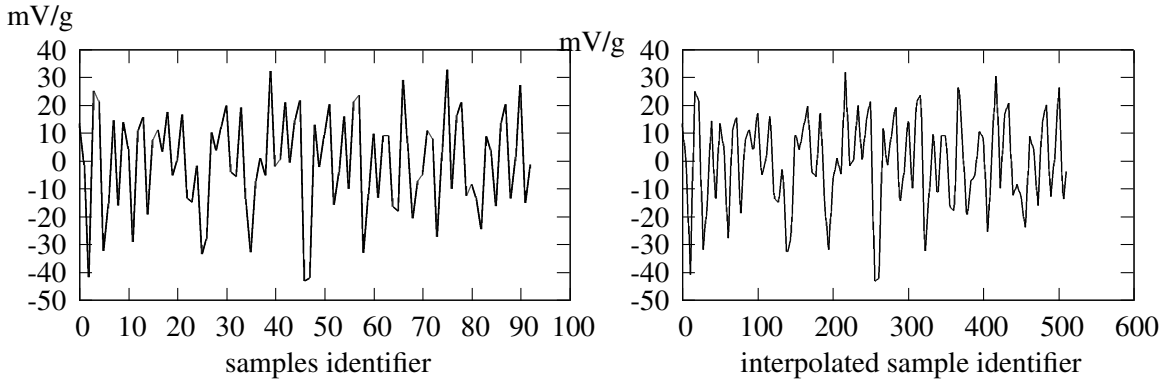
Like the first revolution, the second one has 92 samples (figure 4.10a) and its vibration level is between -40 and 30mV/g. This signal is interpolated on 512 points and produces a signal with the same shape and the same level of vibration (figure 4.10b).



(a) raw vibration signal of the 1st monitored shaft revolution

(b) Interpolation of the first monitored shaft revolution

Figure 4.9: raw vibration signal of the 1st monitored shaft revolution and its interpolation



(a) raw vibration signal of the 2nd monitored shaft revolution

(b) Interpolation of the 2nd monitored shaft revolution

Figure 4.10: raw vibration signal of the 2nd monitored shaft revolution and its interpolation

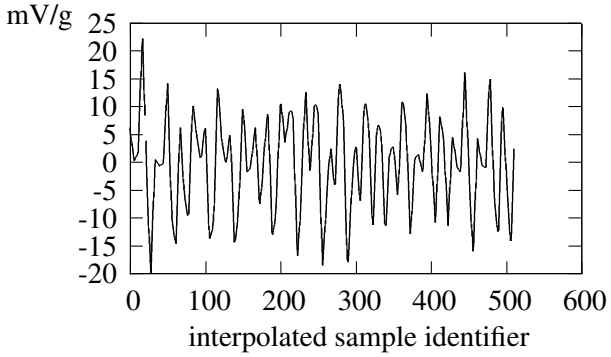


Figure 4.11: Average of all monitored shaft revolutions after interpolation

4.7 Computing a synchronous average of all monitored shaft revolutions

The acquisition file has 31 reference shaft revolutions (*Ntop* = 32 and r_t = 16.82). According to equation 3.2

$$Nsim = \lfloor (Ntop - 1) \times r_t \rfloor = 521$$

This experiment consists in averaging all the monitored after interpolation. The signal represented in figure 4.11 is obtained by averaging all the *Nsim* interpolated revolution of the monitored shaft. The vibration level is between -50 and 40mV/g.

We notice that the vibration level measured at the *tail drive shaft 1* is between -50 and $40mV/g$ during the acquisition phase. This is observable on the vibration signals of the two first monitored shaft revolutions and on the averaged signal. The most important thing to note in this example is the fact that averaging the vibration signals of all monitored shaft revolutions of the monitored shaft has a meaning since they have the same shape and the same level of vibration; thus reduces the noise.

4.8 Computing HMS on-ground indicators

The existing HMS function computes frequency and temporal indicators. In the following experiments, we compute 2 frequency indicators (OM1, OM2) and 2 temporal indicators (RMSb, RMSR) using an acquisition file that contains two accelerometers that monitor respectively the *input drive shaft 1*.

4.8.1 Frequency indicators

OM1 indicator

The OM1 indicator is calculated as follows:

$$\mathrm{OM1} = \overbrace{2 \times \overbrace{\mathrm{SelectHarmonic}(\underbrace{\mathrm{Module}(\underbrace{\mathrm{FFT}(\mathrm{AverageSignal})}_{1})}_{2}, 1)}^{3}}^{4}$$

where AverageSignal represents the average signal of all *Nsim* monitored shaft revolution. We define that

$$\mathrm{fft} = \mathrm{FFT}(\mathrm{AverageSignal})$$

$$\text{Module(fft,i)} = \sqrt{(Re(\text{fft}[i])^2 + Im(\text{fft}[i])^2)}$$

The OM1 indicator is twice the amplitude of the rotation frequency of the monitored shaft. Recall, it consists in calculating first the FFT of the AverageSignal. The FFT function is implemented in A.6 using the *FFTW* library. According to [26], the *FFTW* library is a C subroutine library providing Discrete Fourier Transform with arbitrary input size. It also supports real and complex data type.

1. In our use-case, the FFT transforms the average signal (AverageSignal) which has IT samples (this array is named *real* in A.6, line 4) into an array of IT complex (named *complex_fft* in A.6, line 4). The usage of FFTW library to compute FFT has four main steps. We first create a *plan* by invoking the function *fftw_plan_dft_1d*. According to [26], a plan is an object that store all the data that FFTW needs to compute the FFT. Only one plan is associated to an array of samples. After configuring the input and output arrays, we define also which FFT we want to compute: a *Forward* Fast Fourier Transform or a *Reverse* Fast Fourier Transform. In others words, we have to define the *sign* of the exponent in the formula 4.5 that calculates the FFT. As a reminder the FFT is calculated as follows:

   $$x(n), n \in [0..N-1]$$

   We define its discrete Fourier transform as the sequence $X(l)$:

   $$X(l) = \sum_{n=0}^{N-1} x(n) W_N^{-nl}, l \in [0..N-1] \tag{4.5}$$

   $$W_N = e^{j2\pi/N}$$

   This sign is equal to -1 in a case of Forward FFT (equation A.6) and +1 when computing a reverse FFT. In the same way, the signs -1 and +1 are respectively replaced in the FFTW library by the flags FFTW_FORWARD and FFTW_BACKWARD (refer to line 12 in Listing A.6). The fourth and last argument to set when creating a *plan* is either the flag FFTW_ESTIMATE or FFTW_MEASURE. According to [26], the flag FFTW_ESTIMATE provides a *reasonable* plan while FFTW_MEASURE finds the optimal plan by computing several FFTs. Then, the next step after creating a plan is to transform each real array of samples AverageSignal into complex array of samples. This transformation step is performed by the function *generate_input_fft* (refer to the 16^{th} line of Listing A.6). The full implementation of this function is illustrated in A.4. It basically transforms an array of IT samples into an array of $2 \times$ IT samples. At the third step, the FFT is computed via *fftw_execute(plan)* and normalized the FFT results by the number of input samples IT. It is very common when computing a Discrete Fast Fourier Transform using a library to ask about which version is implemented: an unnormalized or a normalized one as illustrated in [27]. According to [26], the FFTW library computes an unnormalized FFT. It means that, computing a forward FFT following by a reverse FFT will produce the original inputs scale by IT.

2. The FFT of the synchronous average signal produces IT complex. Then, the module of the Fast Fourier Transform (refer to Listing A.5) calculates the amplitude of the harmonics of the spectrum. In addition, $module[0]$ is called the fundamental, the first harmonic ($module[1]$) is positioned in the spectrum at the frequency fs, the second harmonic ($module[2]$) is $2 \times$ fs, the third harmonic ($module[3]$) is $3 \times$ fs and so on.

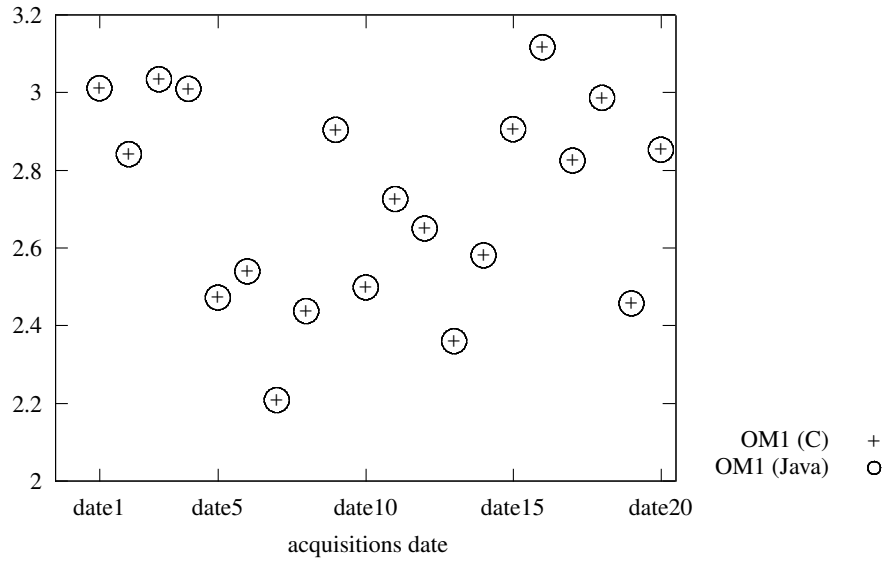


Figure 4.12: Comparison of on-ground OM1 indicator between C and Java versions

3. The third step consists in selecting the 1st harmonic of the spectrum. Following up the nomenclature of the HMS indicators, OM1 is the first harmonic ($module[1]$), OM2 the second harmonic ($module[2]$) and so on.

4. Finally, the amplitude of the spectrum is multiplied by 2 because the FFT of a real signal is symmetric.

Figure 4.12 shows the value of the OM1 indicator calculated for several acquisition files. In this test, we consider 20 acquisition files and the monitored shaft is the *input drive shaft 1* which has a rotational speed ratio r_t = 16.82 with respect to the tail rotor. The x-axis represents the acquisition dates and the y-axis represents the value of OM1 indicator in g. For each acquisition file, one OM1 is calculated by first using the *C* algorithm then the existing Java version. For all acquisition files, the OM1 calculated with the 2 versions are strictly equal. In addition, the main potential source of difference between the *C* and *Java* versions we have investigated is the interpolation step. In fact, we have designed our own *linear interpolation* function based on mathematical theory while the *Java* version used a *polynomial spline function* from *apache* library. The polynomial spline function is used as a linear interpolation function that takes as argument linear coefficients. We have performed unit test experience to compare these two types of implementation of a linear interpolation function. For the same input data (AverageSignal), the interpolated signal produced by these 2 types of implementation is the same.

OM2 indicator

The OM2 indicator is calculated as follows:

$$\mathrm{OM2} = \overbrace{2 \times \overbrace{\mathrm{SelectHarmonic}(\underbrace{\mathrm{Module}(\underbrace{\mathrm{FFT}(\mathrm{AverageSignal})}_{1})}_{2}, 2)}^{3}}^{4}$$

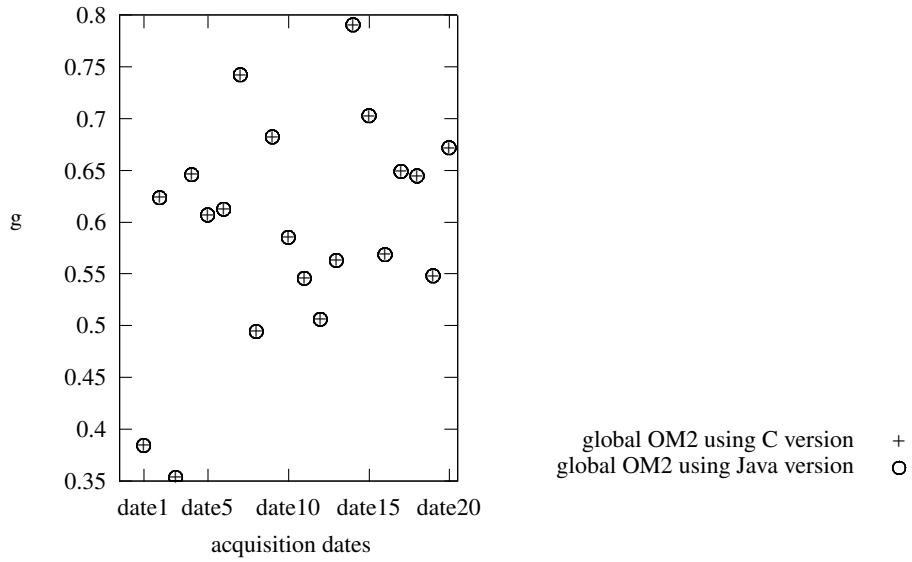


Figure 4.13: Comparison of on-ground OM2 indicator between C and Java versions

where AverageSignal represents the average signal of all *Nsim* monitored shaft revolutions. OM2 is twice the amplitude of the second harmonic of the module of spectrum of the AverageSignal signal ($module[2]$).

Like the OM1 indicator, the OM2 indicator is computed into four steps. It consists in calculating first the FFT of the AverageSignal signal using the FFTW libray (refer to Listing A.6). Then the module of the Fast Fourier Transform is computed using the synchronous average signal as implemented in A.5. The third step consists in selecting the 2nd harmonic of the spectrum. Finally, the amplitude of the spectrum is times 2 because the FFT of a real signal is symmetric.

Figure 4.13 shows the value of the OM2 indicator calculated for 20 acquisition files and the monitored shaft is the *input drive shaft 1* which has a rotational speed ratio r_t = 16.82 with respect to the tail rotor. The x-axis represents the acquisition dates and the y-axis represents the value of OM2 indicator in g. For each acquisition file, one OM2 is calculated on vibration data of the *input drive shaft 1* by first using the *C* algorithm then the existing *Java* version. For all acquisition files, the OM2 calculated with the 2 versions are strictly equal.

4.8.2 Temporal indicators

RMSb indicator

The RMSb indicator is calculated as follows:

$$\mathrm{RMS}(\mathrm{rawSignal})$$

where rawSignal is the raw vibration samples measured by the accelerometer for a given monitored shaft. Unlike previous indicators OM1 and OM2 which are computed on the average signal (AverageSignal), the indicator RMSb is computed using the raw vibration signal. *There is no need to detect the reference and monitored shaft tops when computing the RMSb indicator*. The implementation of the RMSb indicator is shown in Listing A.7 and according to the equation 2.1. The RMSb indicator is used in order to evaluate the energy dispersion of the raw signal. Given an array of samples, it is equal to the square root of the variance. In addition, we need to know the mean to evaluate the variance. That is the reason

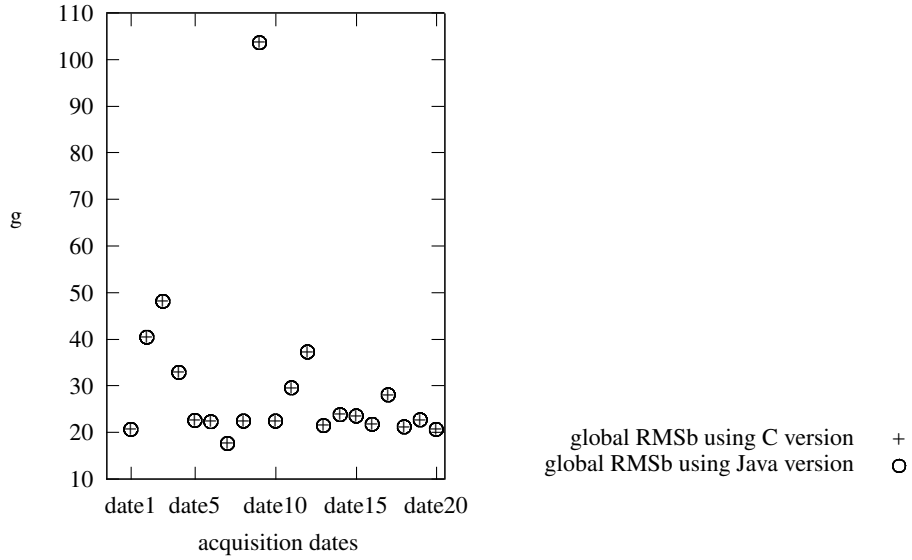


Figure 4.14: Comparison of on-ground RMSb indicator between C and Java versions

why, in Listing A.7, the RMSb indicator of the rawSignal is calculated into 3 steps: we first call the function *computeMean*, then follow by *computeVariance* and finally the square root of the variance (line 22, Listing A.7).

Figure 4.14 depicts the value of the RMSb indicator calculated for 20 acquisition files and the monitored shaft is the *input drive shaft 1* which has a rotational speed ratio r_t = 16.82 with respect to the tail rotor. The x-axis represents the acquisition dates and the y-axis represents the value of RMSb indicator in g. For each acquisition file, one RMSb is calculated by first using the C algorithm then the existing Java version. For all acquisition files, the RMSb calculated with the 2 versions are strictly equal.

RMSR indicator

The RMSR (Root Mean Square Residual) indicator is calculated as follows:

$$\overbrace{\text{RMS}(\overbrace{\text{FFT}^{-1}\left(\underbrace{Remove(\underbrace{\text{FFT}(AverageSignal)}_{1},V)}_{2}\right)}^{3})}^{4}$$

$AverageSignal$ denotes the average of all monitored shaft revolutions and V is the set of harmonics to remove from the spectrum.

The computation of the RMSR indicator can be divided into four parts. The implementation of this indicator is shown in Listing A.8.

- The first part consists in calculating the Fast Fourier Transform using the AverageSignal.
- During the second step, harmonics specified in the V argument are removed. The function *residualfftComplex* in A.9 allows to remove a set of harmonics from the spectrum. It has as inputs four parameters: the complex spectrum on which we want to remove a set of harmonics (*fft*), its size (*fft_length*), the vector of harmonics to be removed from the spectrum and their multiple indices (*lines*) and the size of this vector (*length_list*).

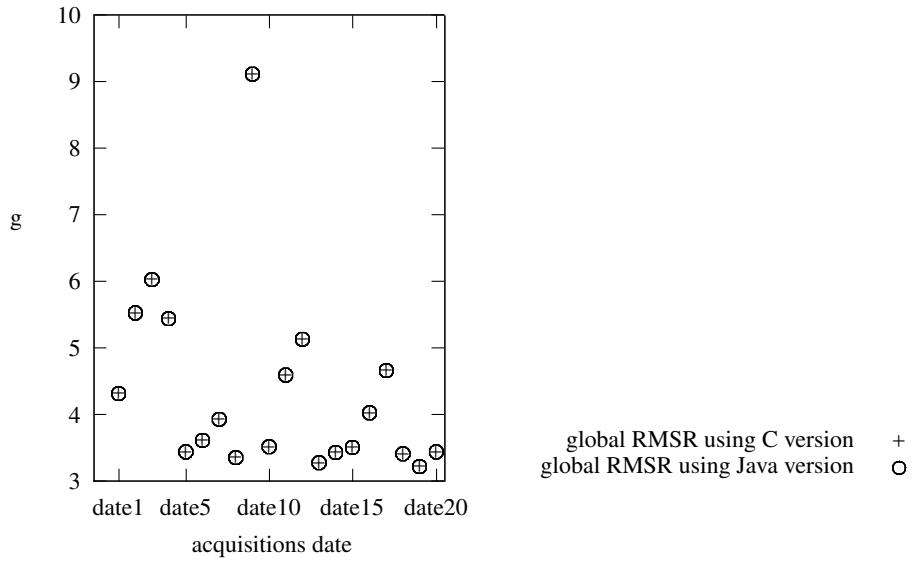


Figure 4.15: Comparison of on-ground RMSR indicator between C and Java versions

- The third step consists in computing the reverse FFT of the residual frequency spectrum. Computing a reverse FFT follows the same steps as the forward FFT except that during the plan creation we must specify the flag *FFTW_BACKWARD* (refer to line 7, Listing A.10).
- Finally, the root mean square (RMS) is calculated using the residual temporal signal. As a reminder, the full implementation of the RMS is done in Listing A.7.

Figure 4.15 shows the value of the RMSR indicator calculated for 20 acquisition files and the monitored shaft is the *input drive shaft 1* which has a rotational speed ratio r_t = 16.82 with respect to the tail rotor. The x-axis represents the acquisition dates and the y-axis represents the value of RMSR indicator in g. For each acquisition file, one RMSR is calculated by first using the C algorithm then the existing Java version. For all acquisition files, the RMSR calculated with the 2 versions are strictly equal.

4.9 Sensitivity of on-ground indicators

Because indicators are associated with monitored shaft revolutions, we must be able to delimit the beginning and the end of each revolution of the monitored shaft.

When estimating the beginning and end of a monitored shaft revolution, we compute the position as an integer index on the time scale of the individual samples (the accelerometers and the phase sensor being sampled at the same frequency). This involves some rounding operations.

The following experiments aim to assess the sensitivity of the indicators to the positions of the monitored shaft tops and the interpolation type.

Indicator sensitivity to the top positions of the monitored shaft

We consider the same acquisition file in the following 3 experiments. The goal is to vary the position of the tops of the monitored shaft and to evaluate the impact on the results of the indicators. This test is done on 3 indicators (OM1, OM2 and RMSR) because for these indicators, we need to delimit the revolutions of each monitored shaft; which is not the case for the temporal indicator RMSb. For each

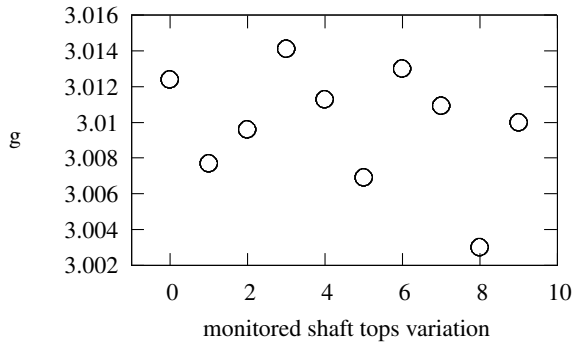


Figure 4.16: OM1 indicator according to the variation of the tops of the monitored shaft

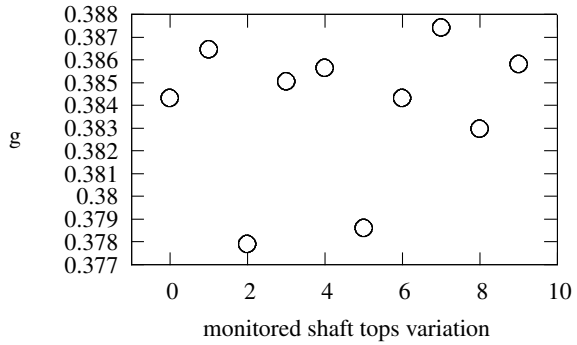


Figure 4.17: OM2 indicator according to the variation of the tops of the monitored shaft

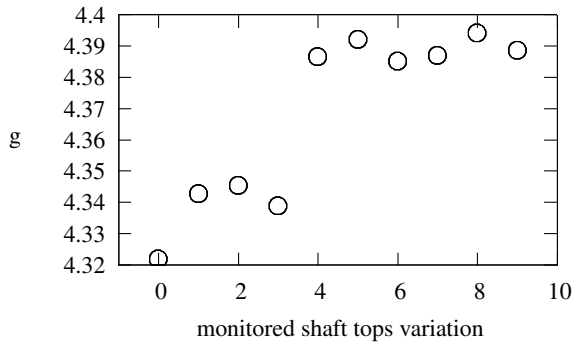


Figure 4.18: RMSR indicator according to the variation of the tops of the monitored shaft

indicator, the monitored tops are varied from 0 to 9 samples. Consequently, the first value of each figure corresponds to the nominal case. The nominal case corresponds to indicators calculated without shifting the position of the tops. The figure 4.16 shows the OM1 indicator by shifting the monitored shaft tops from 0 to 9 samples. Note that the OM1 calculated in nominal case is equal to $3.012429g$ while it is between 3.006927 and $3.014147g$ after shifting the tops between 1 and 9 samples. This gives a maximum variation of 0.05%.

The same experiment was performed to calculate the OM2 and RMSR indicators. The results of these experiments are illustrated by the figures 4.17 and 4.18. The OM2 indicator is equal to $0.384336g$ in the nominal case and the maximum difference obtained is 0.55%. Concerning the RMSR, it is equal to $4.322097g$ in the nominal case and has a maximum variation of 1.65% by shifting the tops from 1 to 9 samples.

This serie of experiments shows that by shifting the position of the monitored tops from 1 to 9 samples, there is a slight variation in the indicators in general. In addition, it is noted that this variation has a greater impact on the temporal indicator than on the frequency indicators.

4.10 Limits of the convergence criterion

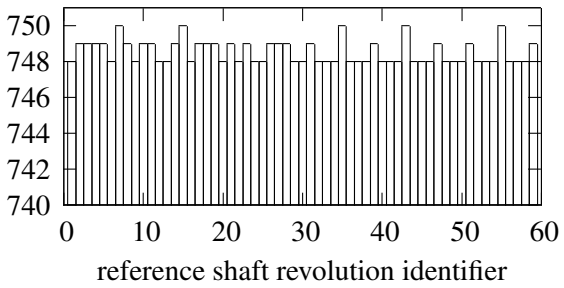


Figure 4.19: Number of samples of the reference shaft per revolution, the speed of the rotor is almost constant

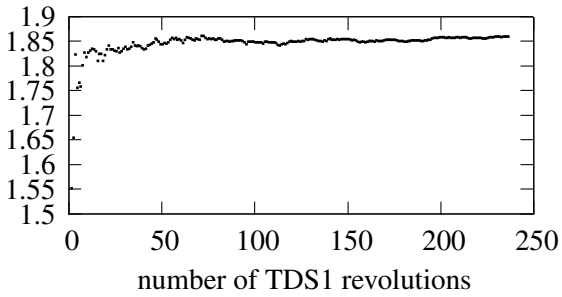


Figure 4.20: Evolution of OM1 indicator according to the number of TDS1 revolutions, the speed of the rotor is almost constant

We consider 2 acquisition files that satisfy the convergence criterion described in the paragraph 3.6 by calculating the covariance between the average synchronous signal for *Nsim* revolutions and average synchronous signal for $\frac{Nsim}{2}$ revolutions of the monitored shaft. Recall, the formula of the covariance is given by the equation 3.5.

The main difference between these 2 files is the conditional operations during the flight. Figures 4.19 and 4.21 depict the number of samples of the tail rotor per revolution during the acquisition session. We note that the speed of the tail rotor is almost constant: the number of samples per revolution varies slightly (between 748 and 750 samples). Nevertheless, the speed of the rotor varies much more in the figure 4.21 and is in the range $\{749, 816\}$. Thus, it basically means that, in the figure 4.21 the speed of the rotor increases during the acquisition session.

Moreover, this variation of the speed of the rotor has an impact on the results of the OM1 indicator. We observe that when the rotor speed is almost constant, the OM1 indicator has the same shape. It varies very little and converges very quickly from the 50th revolution of the tail rotor. On the other hand, it increases with the speed of the rotor (figure 4.22). Thus, the OM1 indicator calculated by considering 59

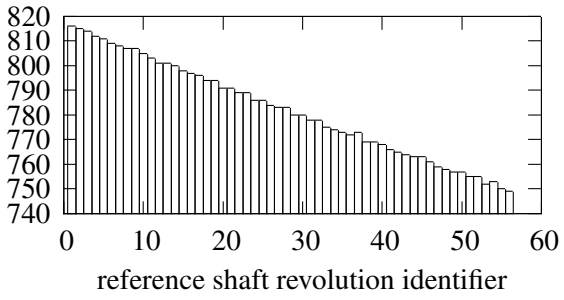


Figure 4.21: Number of samples of the reference shaft per revolution, the speed of the rotor increases

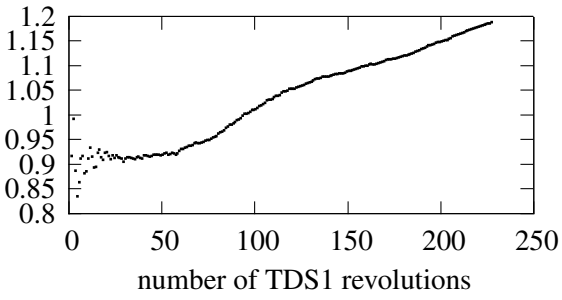


Figure 4.22: Evolution of OM1 indicator according to the number of TDS1 revolutions, the speed of the rotor increases

revolutions of the tail rotor would be different if we had increased the acquisition window. Such a value of OM1 will be much higher than its true value (i.e OM1 linked only to defect of the monitored shaft) and will lead to false alarms.

This experience shows that the convergence criterion is insufficient. The parameters of the helicopter such as the speed of the rotor should be taken into account to compare indicators from data acquired under the same flight conditions. In addition, the parameters related to only the machine will not be sufficient, the flight environment like the air speed must be considered to build a rigorous model.

4.11 Conclusion

In this chapter, we first highlight the helicopter flight regime by calculating the speed of the rotor during the acquisition session. We notice that the acquisition sessions are not performed during a strict stationary regime of the helicopter. There are files in which we observe a regular acceleration of the rotor (case 3). This variation makes the interpretation of the indicators more challenging because we compare the trend of a given indicator which vibration samples are acquired under the same conditions. The article [28] describes the principle of detection of mechanical defection of the *Gazelle Helicopter SA-342*. The frequency spectrum of the monitored shaft is used to analyze the vibrations. Experiments have shown that by comparing the frequency spectrum of the monitored shaft in a phase of flight without acceleration (reference version) to the spectrum in which the pilot is accelerating, the spike at $60.45Hz$ has increased by at least 30%. In our case study, since it is impossible to perform acquisitions at a given speed with rigorously the same flight conditions, another filtering criterion called *convergence criterion* is used to filter the acquired data before calculating the indicators. However, the experiment in case 3 satisfies the convergence criterion. It means that this criterion is insufficient because it aims to filter data acquired during a regular acceleration or deceleration. In addition, increasing the value of the threshold $limit$ to which the covariance CONV is compared will not solve the problem. The vibration samples must be associated with helicopter contextual parameters.

After filtering the vibration data by using the SFL flight regime and the convergence criterion, then at each stage of the computation we compare the results obtained with two different ways. These stages are: the reference shaft top detections, dividing a reference shaft into monitored revolutions knowing the speedup ratios, interpolating each monitored shaft revolutions and finally the average signal.

The frequency and temporal indicators obtained with the reference C version and the existing Java

one are the same even if the implementation of theses two versions are different. For instance, we have implemented a linear interpolation function without using any external library.

In the following chapter, we will illustrate how to transform our C reference on-ground global HMS into an on-ground incremental version. Our C on-ground reference version needs the full data set to compute indicators. This incremental version should be able to produce incremental results.

Chapter 5

Building an Incremental Version of the HMS Algorithm

In the previous chapter, we have shown that, given the same input files, our *C reference version* produces the same indicators as the Java existing one. This chapter focuses on the transformation of our C reference version into an *incremental one*.

We have around 80 raw acquisition files to compare the new versions with the reference one. These files cover all the different acquisition categories (refer to 2.4). The maximum number of acquisition files of a given category is 33.

First, we will focus on input data from the acquisition card to the processing unit. The incremental algorithm should be able to treat data as soon as they become available, because it is impossible to store them before computing the indicators. The management of the inputs flow (vibration samples and tops) aims at addressing the data availability in embedded processing device and when they are consumed to compute indicators.

Second, we need to transform our C reference version. Because, with the C reference algorithms, we consider the full data set to compute indicators. For instance, we have shown that in Algorithm 3.2, page 39, the current HMS uses the global average *monitored_revolution_mean*. This global average is the average of all monitored shaft revolutions between the first and the last detected reference tops in the acquisition file. It is used to estimate the positions of the tops of the monitored shaft. However, this is not feasible for systems that have to work on the flow of inputs in real-time. The global average *monitored_revolution_mean* has to be transformed into an incremental one.

Finally, we use a HOST PC to simulate the input flow.

5.1 Problem Statement

5.1.1 Input flow

In contrast to the existing approach of the HMS which requires to gather all the vibration data in an acquisition file before computing indicators, the incremental computation refers to a computation which is updated as soon as data are available. The on-ground (reference) HMS no longer satisfies this constraint due to lack of memory to store all the vibration data.

Note that the health indicators are not evaluated for each new vibration sample. The computation is updated every time a new vibration *chunk* arrives (a chunk is a packet of samples, its size is defined by the latency experiment in Section 8.2.2, page 113). The computation of indicators is synchronized with

the tops of the phase sensor. The number of tops inside a packet of vibration samples is variable because the speed of the rotor varies. This leads to burst processing.

5.1.2 Transforming the C reference Algorithm

An effort is needed to transform and re-adapt the reference implementation into an incremental version. The incremental version updates the computation for newly available vibration samples. We have also tracked all the C reference implementation by looking for blocking points that could make this transformation hard. The main blocking point is the use of the global average *monitored_revolution_mean* to estimate the tops of the monitored shaft. The transformation of the reference algorithms will necessarily introduce some errors with respect to our reference version. We have identified two sources of errors.

First, the error comes from the use of *monitored_revolution_mean* to estimate the beginning and the end of each monitored shaft revolution. We use two different strategies to replace the global average (*monitored_revolution_mean*) by an incremental average using either a *growing window* or a *sliding window*.

Second, the difference is also observed for indicators that do not involve the estimation of the tops of the monitored shaft such as RMSb. In the former case, the error comes from the loss of precision in our implementation by accumulating errors.

5.2 The incremental computations: management of inputs

For the sake of simplicity, we present the flow of inputs of an on-ground incremental version of the health indicator considering the simple case in which there is a single phase sensor (on the main rotor), and a single accelerometer, on a rotating shaft which has a speedup ratio 4 with respect to the main rotor. In addition, we replace also for clarity our notation in Figure 5.1: REF_REVOLUTION$[i]$ by NR$[i]$. This is enough to show the management of inputs with the samples arriving in packets. The indicators will not be updated for every new single vibration data because vibrations are associated with monitored shaft revolutions.

More complex cases (several sensors, or non-integer speed ratios), are refinements of this presentation. When several sensors are used, we have to perform transpositions on the flow of inputs.

Figure 5.1 illustrates how the flow of inputs is read in chunks, and how its samples are then grouped according to the position of the tops. It also shows the earliest time at which the grouped data are available for further treatments.

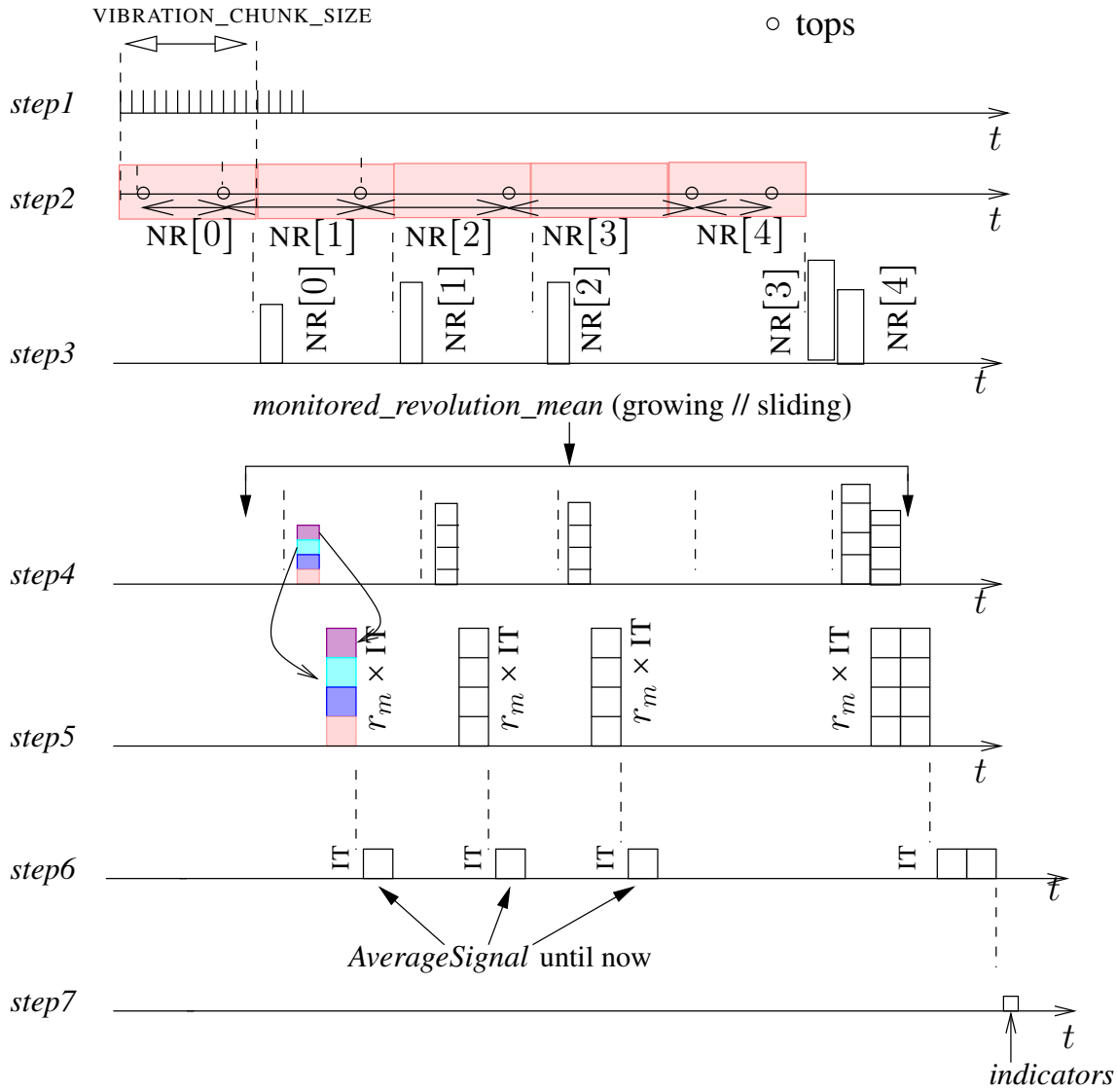


Figure 5.1: Step1 is the vibration and top samples, step2 corresponds to the packet transmission from acquisition unit to the processing device, step3 gathers samples per reference revolution, step4 estimates the tops of a monitored shaft with a speedup ratio r_m using either a growing or a sliding window, step5 is the interpolation step with IT samples, step6 calculates the average signal "*until now*" and finally step7 computes the indicators.

Step1 represents the discrete samples, taken at a constant sampling frequency fs. The sampling frequency is in the range $[1 - 31]kHz$. Recall, the acquisition sessions are performed when the speed of the rotor is in a specific range: [SPEED_RPM_MIN, SPEED_RPM_MAX] and NR is the nominal speed of the rotor (in revolution per minute); NR is a constant.

$$\text{SPEED\_RPM\_MIN} = 50\% \times \text{NR}$$

and

$$\text{SPEED\_RPM\_MAX} = 75\% \times \text{NR}$$

Step2 illustrates the transmission of packets of size VIBRATION_CHUNK_SIZE from the acquisition

unit to the processing device. Packets are composed of samples of shafts to be monitored, and tops of the reference shaft (denoted by small circles). Since the speed varies, the number of tops in a given packet varies. For instance, the first packet of samples has two tops, while the second one has only one top and there may be also packet which have no top like the fourth packet.

On step3 we gather samples per revolution (i.e., data between 2 tops). The data is available a bit later than the end of the packet. The vertical rectangle is an array of samples of size $\text{NR}[0]$, then $\text{NR}[1]$, etc. We note that the height of each array is variable due to the variation of the speed of the reference shaft. Notice that $\text{NR}[3]$ and $\text{NR}[4]$ are available late, and at the same time, due to the absence of tops in the fourth packet of size VIBRATION_CHUNK_SIZE.

One packet of size VIBRATION_CHUNK_SIZE can represent 0 to K complete revolutions of the reference shaft. K is bounded by

$$\text{VIBRATION\_CHUNK\_SIZE} / \left(\frac{\text{fs}}{60 \times \text{SPEED\_RPM\_MAX}} \right)$$

where fs is the sampling frequency (*Hz*) and SPEED_RPM_MAX is the maximum rotational speed of the reference shaft (given in revolutions/minute) during the acquisition session;

At step4 each array of samples $\text{NR}[i]$ is split into r_m equal parts. Since the speed varies between two tops of the phase sensor (i.e., $\text{NR}[i] \neq \text{NR}[i+1]$), and r_m is not always an integer, there will be sets of samples corresponding to one revolution of the monitored shaft, which spread across a top. In the current reference implementation, the interpolated points are placed according to the global speed average. This is clearly not feasible in the incremental version, where we use either an incrementally computed *average until now* (growing window) or *sliding average* (sliding window), instead. The growing and sliding average are discussed in the next section.

Then step5 is the interpolation phase: when an array $\text{NR}[i]$ is available, it represents $r_m = 4$ revolutions of the monitored shaft. For each of them, we produce a fixed number "IT" of samples, by linear interpolation, assuming that the speed of the reference shaft is *constant* during *one* revolution. We now have arrays of $r_m \times \text{IT}$ points.

At step6, the synchronous average signal is computed. For each new interpolated monitored shaft revolution, the synchronous average is updated. This step requires only one array of IT samples. This array has two objectives: firstly to update the average signal for each new interpolated monitored shaft revolution; and secondly to update the average signal for each new packet of vibrations chunk coming from the acquisition unit thanks to the *growing* principle (will be detailed in section 5.6).

Finally, the computation of the indicators starts at step7 once the synchronous signal is already calculated.

5.3 Growing and Sliding Window Principle

We consider a sequence of N values: $\text{x}(i), i \in 1..N$. The growing and sliding window principles are respectively illustrated in figures 5.2 and 5.3 with a sequence of 5 values. In our case study, $\text{x}(i)$ is the mean number of samples of the i^{th} reference shaft revolution. These principles can be extended to array of samples.

5.3.1 Growing Window

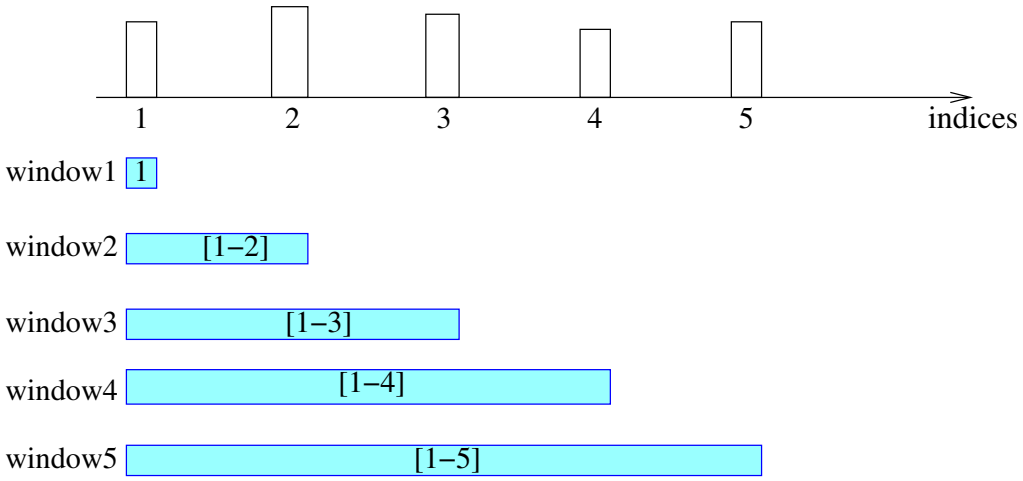


Figure 5.2: Growing Window principle with a sequence of 5 values

Figure 5.2 illustrates the growing window principle. Window1 allows to compute the mean number of samples of the first monitored shaft revolution. Window2 gathers the two first mean number of samples. At this step, the mean is calculated by averaging the two first means. In the same way, window3 computes the mean number of samples of the 3 previous revolutions of the monitored shaft. In general, the mean number of samples of the monitored shaft revolution calculated at step k using a growing window are obtained by averaging all the k previous monitored shaft revolutions.

$$m_k = \frac{\sum_{i=1}^{k} \mathrm{x}(i)}{k}$$

The growing window allows to gradually compute the mean number of samples of the monitored shaft revolution by taking into account all the history. However, the term "growing" does not mean that we constantly need more memory for the computation. The mean m_{k+1} can be calculated incrementally with a loss of precision due to operations involving floating values.

$$m_{k+1} = m_k + \frac{\mathrm{x}(k+1) - m_k}{k+1}$$

5.3.2 Sliding Window

Figure 5.3 illustrates the sliding window principle. Unlike the growing window, the sliding window does not consider the full history but only a *subset* of N values called window size. In figure 5.3, the window size is equal to 3 and the sliding step is 1.

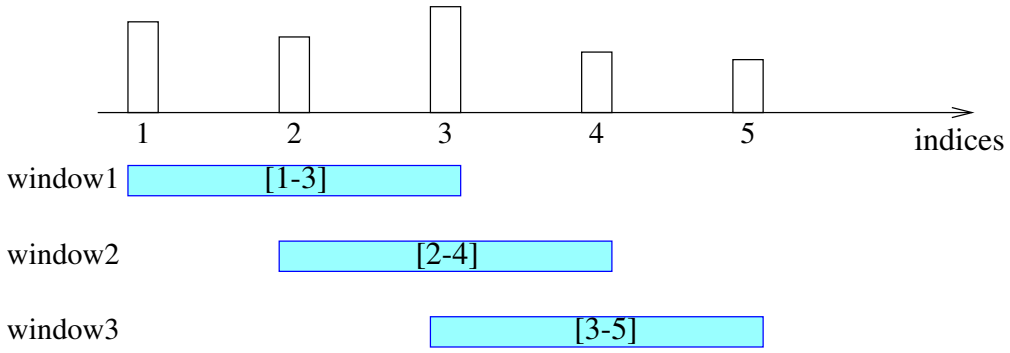


Figure 5.3: Sliding Window principle with a sequence of 5 values. The window size is N=3 and the sliding step is 1.

The mean number of samples of a monitored shaft at step k is given by:

$$x_k = \frac{\sum_{i=k-(N-1)}^{k} \mathrm{x}(i)}{N}$$

We have to store N values corresponding to the window length to compute the sliding mean.

5.4 Principle of an incremental computation: case of OM1 indicator

This experiment shows the evolution of the OM1 indicator computed using vibration data coming from the *Tail Drive Shaft 1*. This file contains *Nsim* = 224 revolutions of the monitored shaft.

Recall, the OM1 indicator is computed as follows (refer to section 2.6, page 27):

$$\mathrm{OM1} = \overbrace{2 \times \overbrace{\mathrm{SelectHarmonic}(\overbrace{\mathrm{Module}(\underbrace{\mathrm{FFT}(\mathrm{AverageSignal})}_{1}),1}^{2})}^{3}}^{4}$$

where AverageSignal represents the average signal of *Nsim* monitored shaft revolutions in the reference version. The reference OM1 is equal to $1.823g$.

Unlike the reference version where only one OM1 is computed by using the average signal of all *Nsim* revolutions, this experiment computes OM1 at each revolution by using a growing window (refer to 5.2). The AverageSignal is also re-evalued for each new monitored shaft revolution. Then, the incremental OM1 at step k is calculated by averaging all the k previous shaft revolutions. OM1 converges from the 50th shaft revolution (k = 50) in Figure 5.4. When k = 50, OM1 is equal to $1.84g$. The relative error of OM1 computed in two different manners (reference versus incremental) is equal to 0.9%.

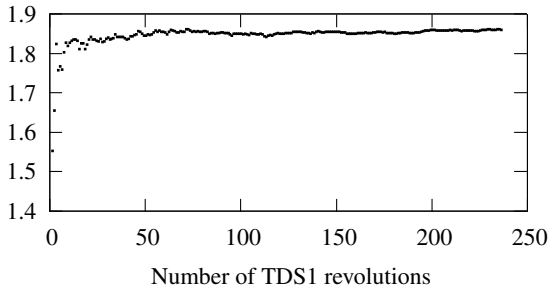


Figure 5.4: Evolution of indicator OM1 according to the number of revolutions of TDS1

This experiment calculates one OM1 value for each new revolution of the monitored shaft TDS1. However, this does not make sense because only the *last* value matters. The value that allows to define the *last* revolution can be configured as a static parameter of the application. This basically corresponds to a number of revolutions at which the indicator is calculated.

Therefore, the incremental implementation no longer computes indicators at each new revolution. But, only the *monitored_revolution_mean* (figure 5.1, step4) and AverageSignal (figure 5.1, step6) are re-evaluated at each new revolution and indicators are calculated at the end after averaging all the *Nsim* revolutions. In others words, given an acquisition file and an indicator, the reference and the incremental implementations produce only one value of that indicator.

5.5 Evaluation criteria for the incremental implementations

When transforming the reference implementation into an incremental one, we need to compare the indicators given by the new version with the reference one. What really matters is the decision taken by the human operator, based on the evolution of the indicators as shown on Fig. 2.9, page 29. The ultimate criteria is that, for the same data, the incremental version leads to the same decisions as the reference implementation. There is no way to decide, based on the algorithm alone, whether the discrepancies produced are acceptable or not. The original designers of the signal processing algorithms have to be consulted, to assess the impact of this change on the results.

However, the incremental indicators will lead to the same maintenance decisions when they are close to the reference ones. In fact, we will compare different incremental versions depending on their proximity to the reference indicators.

Since we know that we can not do exactly like the reference implementation, we try two incremental versions based on the strategy we have adopted to compute *monitored_revolution_mean*:

- with a *growing* average speed (will be detailed in section 5.6)

- with a *sliding* average speed (will be discussed in 5.7)

5.6 Comparing incremental growing window and reference indicators

5.6.1 OM1 indicator

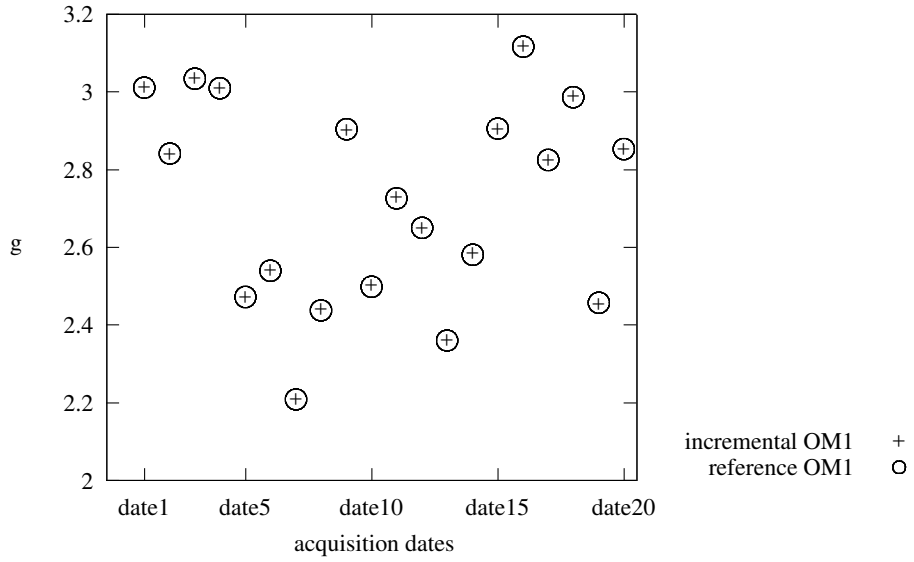


Figure 5.5: Comparison between reference and incremental OM1 indicator using a growing window.

In this experiment, we monitor the shaft *input drive shaft left*, which rotational speed ratio is r_t = 16.82. The purpose of this experiment is to compare the OM1 indicator in two cases. The first case corresponds to the OM1 indicator calculated using the reference implementation (recall, a single value of OM1 is calculated after interpolating and averaging all the monitored shaft revolutions). The second case is the incremental version, one OM1 is also calculated at the end but the main difference is the fact that *monitored_revolution_mean* is updated for each new reference shaft revolution. The average *monitored_revolution_mean* is calculated according the growing window principle.

The experiment is performed on 20 raw vibration files acquired during the same flight conditions. Figure 5.5 shows the OM1 indicator computed using different vibration files with the reference and incremental implementations. The reference and incremental OM1 computed at date1 are respectively equal to $3.013763g$ and $3.012429g$. The two OM1 values are not exactly the same but very close. Finally, the maximum relative difference (reference versus incremental) after computing OM1 indicator using 20 raw vibration files is equal to 0.03% and is observed at date10.

5.6.2 OM2 indicator

The following experiment is performed on vibration data of the monitored shaft: *input drive shaft left* with a speedup ratio equal to r_t = 16.82. Its goal is to compare the OM2 indicator in two cases: the reference implementation and the incremental OM2 indicator calculated using the growing window principle.

Recall, the OM2 indicator is computed as follows (refer to section 2.7, page 28):

$$\overbrace{\mathrm{OM2} = 2 \times \overbrace{\mathrm{SelectHarmonic}(\underbrace{\mathrm{Module}(\underbrace{\mathrm{FFT}(\mathrm{AverageSignal})}_{1})}_{2}, 2)}^{3}}^{4}$$

where AverageSignal represents the average signal of *Nsim* monitored shaft revolutions in the reference version.

We replace the reference AverageSignal of the reference HMS by the average signal "*until now*".

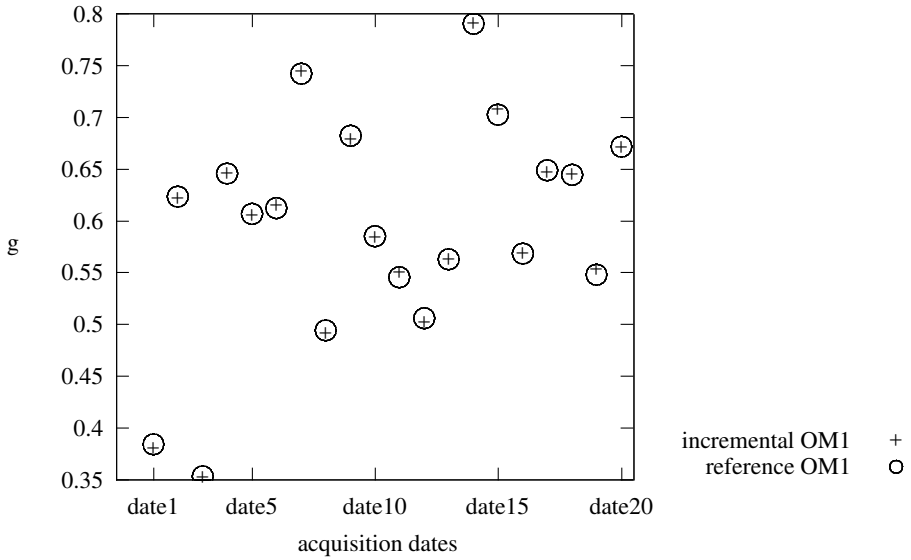


Figure 5.6: Comparison between reference and incremental OM2 indicators using the growing window principle.

The experiment is performed on 20 raw vibration files acquired during the same flight conditions. Figure 5.6 shows the evolution of the OM2 indicator computed using different vibration files with the reference and incremental algorithms. For instance, the reference and incremental OM2 computed at date1 are respectively equal to $0.380728g$ and $0.384336g$. The two OM2 values are not exactly the same but very close. Finally, the maximum relative difference after computing OM2 indicator using 20 raw vibration files is equal to 0.21%.

5.6.3 RMSR indicator

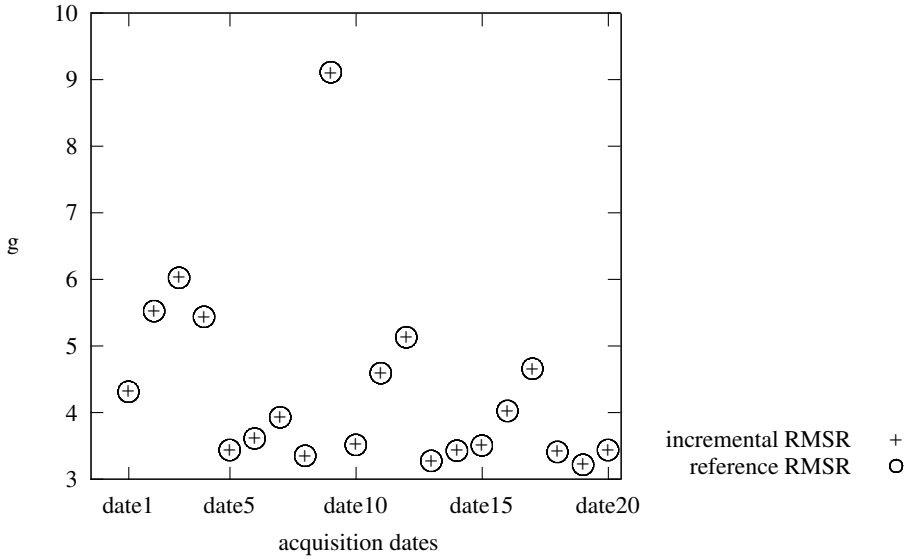


Figure 5.7: Comparison between reference and incremental RMSR indicators using a growing window.

The RMSR (Root Mean Square Residual) indicator is calculated as follows (refer to section 2.4, page 26):

$$\overbrace{RMS(\overbrace{FFT^{-1}\,(Remove(\underbrace{FFT(AverageSignal)}_{1},V))}^{3})}^{4}\Big)}^{2}$$

$AverageSignal$ denotes the average of all monitored shaft revolutions and V is the set of harmonics to remove from the spectrum. We replace the reference AverageSignal of the existing HMS by the average signal "*until now*". The AverageSignal is evaluated using virtual tops of the monitored shaft which are estimated using incremental average *monitored_revolution_mean* thanks to the growing window principle.

This experiment has the same parameters as the previous ones (same acquisition files and monitored shaft) and the purpose is to compare the RMSR indicator calculated using the reference and the incremental algorithms. Figure 5.7 shows that RMSR indicator calculated for 20 raw acquisition files. The maximum difference is observed at date10 and equal to 0.15%.

5.7 Comparing incremental sliding window and reference indicators

5.7.1 OM1 indicator

We use 20 acquisition files that contain vibration samples of the shaft: *input drive shaft left*. The goal of this experiment is to compare the sliding OM1 to the reference one. The sliding indicator is computed using the sliding window principle to evaluate *monitored_revolution_mean*. We consider a window of size 20 reference shaft revolutions.

Figure 5.8 shows the evolution of the OM1 indicator. The maximum difference between the OM1 indicator calculated in two different manners is observed at date 20 and is equal to 0.1%.

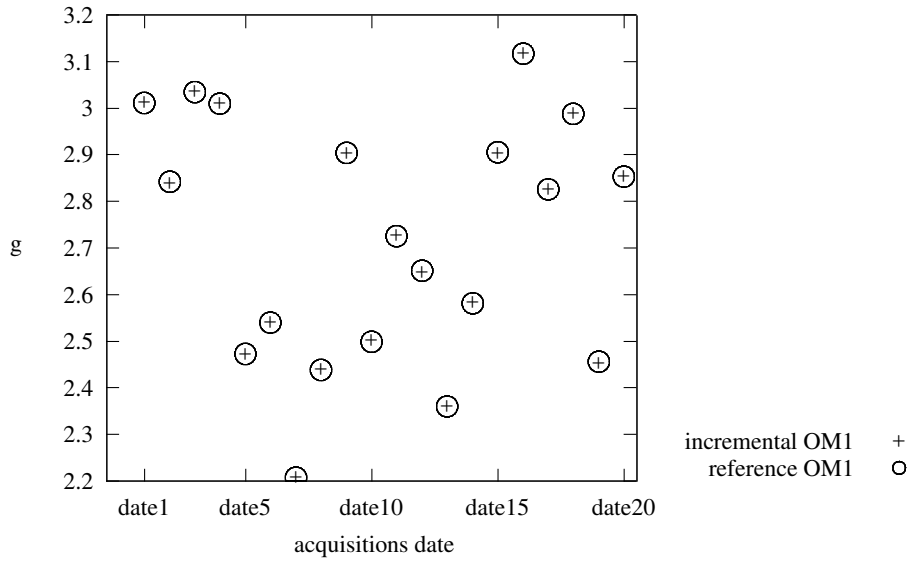


Figure 5.8: Comparison between reference and incremental OM1 indicator using a sliding window. The window size is 20 reference shaft revolutions.

5.7.2 OM2 indicator

In the same way, this experiment aims to compare the sliding OM2 indicator to the reference one. The monitored shaft and the sliding window size are also the same as the previous experiment.

Figure 5.9 shows the evolution of the OM2 indicator. The maximum difference between the two OM1 indicators calculated in two different manners is observed at date 11 and is equal to 0.9%.

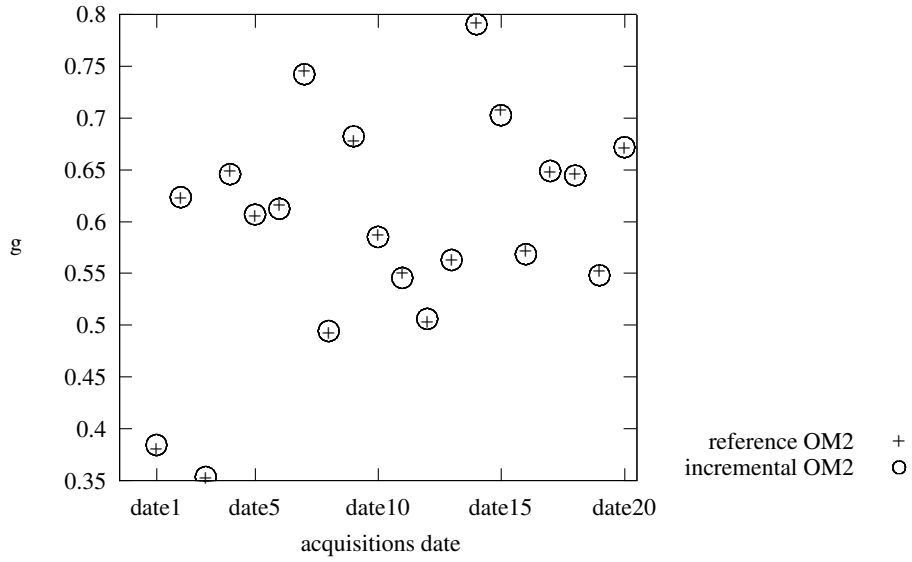


Figure 5.9: Comparison between reference and incremental OM2 indicator using a sliding window. The window size is 20 reference shaft revolutions.

5.7.3 RMSR

This experiment concerns the RMSR indicator and aims to compare the reference RMSR to the sliding one. Figure 5.10 shows the evolution of the RMSR indicator. The maximum difference between the two RMSR indicators calculated in two different manners is observed at date 11 and is equal to 1.5%.

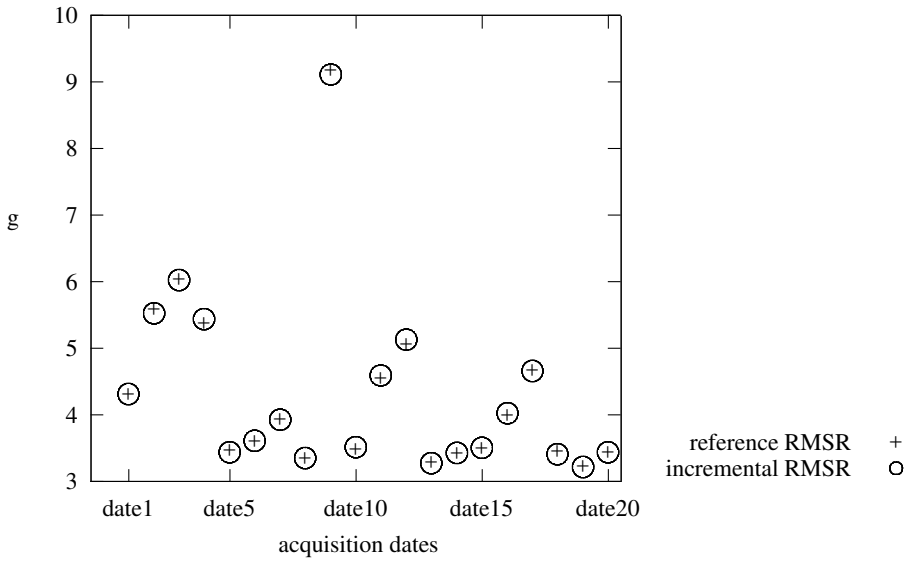


Figure 5.10: Comparison between reference and incremental RMSR indicator using a sliding window. The window size is 20 reference shaft revolutions.

5.8 Comparison between indicators calculated using a sliding and a growing window

This section presents a comparative study between the two incremental indicators implementations calculated using either a sliding or a growing speed. The decision criterion is the following: which of the two incremental versions is closer than the reference one. The relative gap (in %) allows to quantify how close is a given incremental implementation is to the reference one.

Table 5.1 shows the comparison between the 2 incremental versions. For a given shaft, the OM1, OM2 and RMSR indicators are first calculated on 20 acquisition files with the incremental version based on a growing speed. For each of these indicators, the maximum difference with the reference version is calculated. When using for instance the *Input Drive Shaft 1*, the maximum difference of the OM1, OM2 and RMSR indicators compared to the reference version are respectively equal to 0.03%, 0.21% and 0.15%. Then, the same experiment is performed using the incremental version based on the sliding speed. The observed differences are respectively 0.1%, 0.9% and 1.5%.

Monitored shafts	**Speedup ratios**	**Interpolation size**	**Growing and Reference (in %)**			**Sliding and Reference (in %)**		
			OM1	**OM2**	**RMSR**	**OM1**	**OM2**	**RMSR**
Input Drive shaft left	r_t = 16.82	512	0.03	0.21	0.15	0.1	0.9	1.5
Input Drive shaft right	r_t = 16.82	512	0.64	1.11	0.3	0.67	1.24	2.18

Table 5.1: Comparative study between indicators calculated using a growing and a sliding windows.

This comparative study is also applied on vibration data of various monitored shafts which have

different speedup ratio such as the *Intermediate shaft left*, *Intermediate shaft right*, *Combined shaft left* and the *Combined shaft right*. All our experiments have shown that the incremental version based on the growing speed is closer than the existing one. This result can be explained by the fact that the global and growing average are based on the same principle, i.e the reference implementation calculates *once* the average speed by taking into account all the history and the growing principle *gradually* computes the average speed using the history "*until now*".

5.9 Incremental RMSb indicator

In addition to the indicators that take into account the estimated tops of the monitored shaft like OM1, OM2, RMSR, there are also indicators that are directly calculated using the raw signal, for instance the RMSb. It does not involve any estimation of the positions of the monitored shaft. It is calculated as follows:

$$\text{RMSb} = \sqrt{\frac{\sum_{i=0}^{Nbs-1}(v[i] - \text{MEAN}_{Nbs})^2}{Nbs - 1}} \tag{5.1}$$

where

$$\text{MEAN}_{Nbs} = \frac{1}{Nbs} \times \sum_{i=0}^{Nbs-1} v[i]$$

and v is an array of *Nbs* vibration samples.

The computation of the RMSb indicator requires to know the mean value MEAN_{Nbs}. We first need to define an incremental version of the reference MEAN_{Nbs} then derive an incremental implementation of the current RMSb (refer to equation 5.1).

Incremental mean The naive mean is calculated as follows:

$$\text{MEAN}_{Nbs} = \frac{1}{Nbs} \times \sum_{i=0}^{Nbs-1} v[i] \tag{5.2}$$

then from 5.2, $\sum_{i=0}^{Nbs-1} v[i] = Nbs \times \text{MEAN}_{Nbs}$

However, the incremental mean should be evaluated without accumulating large sums.

$$\text{MEAN}_{Nbs} = \frac{1}{Nbs} \times \left(\underbrace{\sum_{i=0}^{Nbs-2} v[i]}_{\text{MEAN}_{Nbs-1} \times (Nbs-1)} + v[Nbs-1] \right)$$

$$\text{MEAN}_{Nbs} = \frac{1}{Nbs} \times \left(\text{MEAN}_{Nbs-1} \times (Nbs - 1) + v[Nbs - 1]\right)$$

The incremental mean at step k requires to know the previous mean and the number of samples.

Incremental Variance

$$\text{RMSb} = \sqrt{\frac{\sum_{i=0}^{Nbs-1}(v[i] - \text{MEAN}_{Nbs})^2}{Nbs - 1}} = \sqrt{\text{VARIANCE}_{Nbs}}$$

We define S_{Nbs} such as:

$$\mathrm{S}_{Nbs} = (Nbs - 1) \times \mathrm{VARIANCE}_{Nbs}$$

The computation of an incremental variance consists of proposing an incremental expression of the term

$$\mathrm{S}_{Nbs} = \sum_{i=0}^{Nbs-1} (v[i] - \mathrm{MEAN}_{Nbs})^2 \tag{5.3}$$

Knuth in [29] proposes a recursive incremental formula to calculate S_{Nbs}:

$$\mathrm{S}_{Nbs} = \mathrm{S}_{Nbs-1} + (v[Nbs - 1] - \mathrm{MEAN}_{Nbs-1}) \times (v[Nbs - 1] - \mathrm{MEAN}_{Nbs}) \tag{5.4}$$

However, his does not show how to derive it from 5.3. The different steps are illustrated in [30] and are the following:

$$\begin{aligned}
\mathrm{S}_{Nbs} - \mathrm{S}_{Nbs-1} &= \sum_{i=0}^{Nbs-1} v[i]^2 - Nbs \times \mathrm{MEAN}_{Nbs}^2 - \\
&\quad \sum_{i=0}^{Nbs-2} v[i]^2 + (Nbs - 1) \times \mathrm{MEAN}_{Nbs-1}^2 \\
&= v[Nbs - 1]^2 - Nbs \times \mathrm{MEAN}_{Nbs}^2 + \\
&\quad (Nbs - 1) \times \mathrm{MEAN}_{Nbs-1}^2 \\
&= v[Nbs - 1]^2 - \mathrm{MEAN}_{Nbs-1}^2 + \\
&\quad Nbs \times (\mathrm{MEAN}_{Nbs-1}^2 - \mathrm{MEAN}_{Nbs}^2) \\
&= v[Nbs - 1]^2 - \mathrm{MEAN}_{Nbs-1}^2 + \\
&\quad Nbs \times (\mathrm{MEAN}_{Nbs-1} - \mathrm{MEAN}_{Nbs}) \times (\mathrm{MEAN}_{Nbs-1} + \mathrm{MEAN}_{Nbs}) \\
&= v[Nbs - 1]^2 - \mathrm{MEAN}_{Nbs-1}^2 + \\
&\quad (\mathrm{MEAN}_{Nbs-1} - v[Nbs - 1])(\mathrm{MEAN}_{Nbs-1} + \mathrm{MEAN}_{Nbs}) \\
&= (v[Nbs - 1] - \mathrm{MEAN}_{Nbs-1}) \times (v[Nbs - 1] - \mathrm{MEAN}_{Nbs}) \\
\mathrm{S}_{Nbs} &= \mathrm{S}_{Nbs-1} + \\
&\quad (v[Nbs - 1] - \mathrm{MEAN}_{Nbs-1}) \times (v[Nbs - 1] - \mathrm{MEAN}_{Nbs})
\end{aligned} \tag{5.5}$$

The incremental variance is implemented in Listing A.11, page 171, according to the formula 5.4.

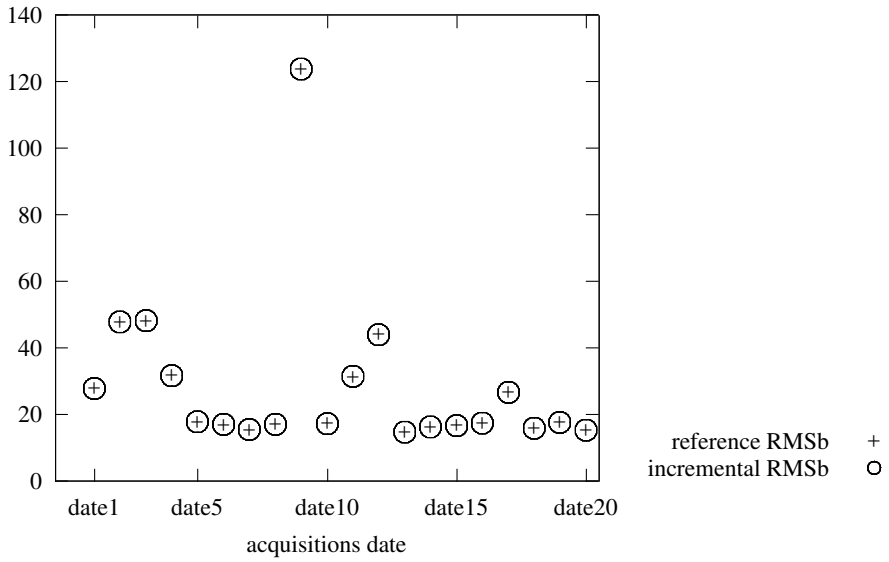


Figure 5.11: Comparison between reference and incremental RMSb indicators using the growing window principle

The incremental RMSb is calculated thanks to the incremental mean and variance implementations. We compare the reference and incremental RMSb indicators calculated using 20 raw acquisition files.

Figure 5.11 shows the evolution of the RMSb indicator. For each file, we plot the incremental and reference RMSb. The minimum and maximum relative errors between these two versions are respectively 0.001% and 0.4%. This difference comes from a loss of precision in our implementation. According to [31], the loss of precision comes from the intermediate division inside the for loop.

5.10 Workload estimation of a packet transmitted by the acquisition unit

This section is devoted to the workload estimation to deal with a packet of samples sent by the acquisition unit to the processing device. As long as we do not specify precisely the mapping of the algorithms on the target architecture, we cannot reason on execution time. We focus on the workload here.

The transition from raw vibration data to functional data used to calculate the indicators goes through 7 steps illustrated in Figure 5.12. We focus on the maximum amount of data to be processed at each step. The idea is that: at each step, we first list the variables and calculate the maximum amount of data to process.

Step2 is the transmission of a packet of size VIBRATION_CHUNK_SIZE from the acquisition unit to the processing device. The size of VIBRATION_CHUNK_SIZE is *constant* and is defined by the latency experiment (will be discussed in the Chapter 8).

Step3 gathers samples per reference shaft revolution. The number of samples per revolution depends on the speed of the rotor and the sampling frequency. Then, the maximum number of samples per reference revolution (NR_{max}) is calculated as follows:

$$\text{NR}_{max} = \frac{\text{fs} \times 60}{\text{SPEED\_RPM\_MIN}}$$

where fs is the sampling frequency (*Hz*) and SPEED_RPM_MIN is the minimum rotational speed of the reference shaft (given in revolutions/minute) during the acquisition session.

However, NR_{max} is not the maximum amount of data at step 3 because there can be 0 to K reference revolutions in a packet of VIBRATION_CHUNK_SIZE samples:

$$K = \text{VIBRATION\_CHUNK\_SIZE} / \left(\frac{\text{fs} \times 60}{\text{SPEED\_RPM\_MAX}} \right)$$

We define Q_{max} as the maximum amount of data at step3 for a packet of size VIBRATION_CHUNK_SIZE:

$$\text{Q}_{max} = \text{NR}_{max} \times K = \frac{\text{SPEED\_RPM\_MAX}}{\text{SPEED\_RPM\_MIN}} \times \text{VIBRATION\_CHUNK\_SIZE}$$

Step4 splits each reference shaft revolution $\text{NR}[i]$ into r_m monitored shaft revolutions when r_m is an integer or up to $\lfloor r_m \rfloor + 1$ when r_m is not an integer speedup ratio. The maximum amount of data is the same as the previous step: Q_{max}.

Step5 is the interpolation step. Each monitored shaft revolution is interpolated into a constant number of samples (IT). Then, we need first to calculate the maximum number of monitored shaft revolutions at this step. For a given packet of vibration samples VIBRATION_CHUNK_SIZE, there can be up to $K \times (\lfloor r_m \rfloor + 1)$ monitored shaft revolutions. Consequently, the maximum amount of data to process at step5 is equal to $K \times (\lfloor r_m \rfloor + 1) \times \text{IT}$ samples.

Step6 computes the synchronous average signal (AverageSignal) until now. The maximum amount of data corresponds to IT samples.

Finally, step7 computes indicators using AverageSignal signal and produces a dozen of indicators (ε) for a given shaft. Each indicator is a float.

All the steps are done at the same time. Therefore, the total workload given: a packet of samples of size (VIBRATION_CHUNK_SIZE), a sampling frequency fs, a monitored shaft with a speedup ratio r_m, an interpolation size IT and finally a rotor with a speed NR is calculated by adding the maximum amount of data of each step.

$$\begin{aligned} \text{W}(\text{VIBRATION\_CHUNK\_SIZE}, \text{fs}, r_m, \text{IT}, \text{NR}, \text{SPEED\_RPM\_MIN}, \text{SPEED\_RPM\_MAX}) = {} & \text{VIBRATION\_CHUNK\_SIZE} \\ & + \text{NR}_{max} \times K \\ & + K \times (\lfloor r_m \rfloor + 1) \times \text{IT} \\ & + \text{IT} \\ & + \varepsilon \end{aligned} \tag{5.6}$$

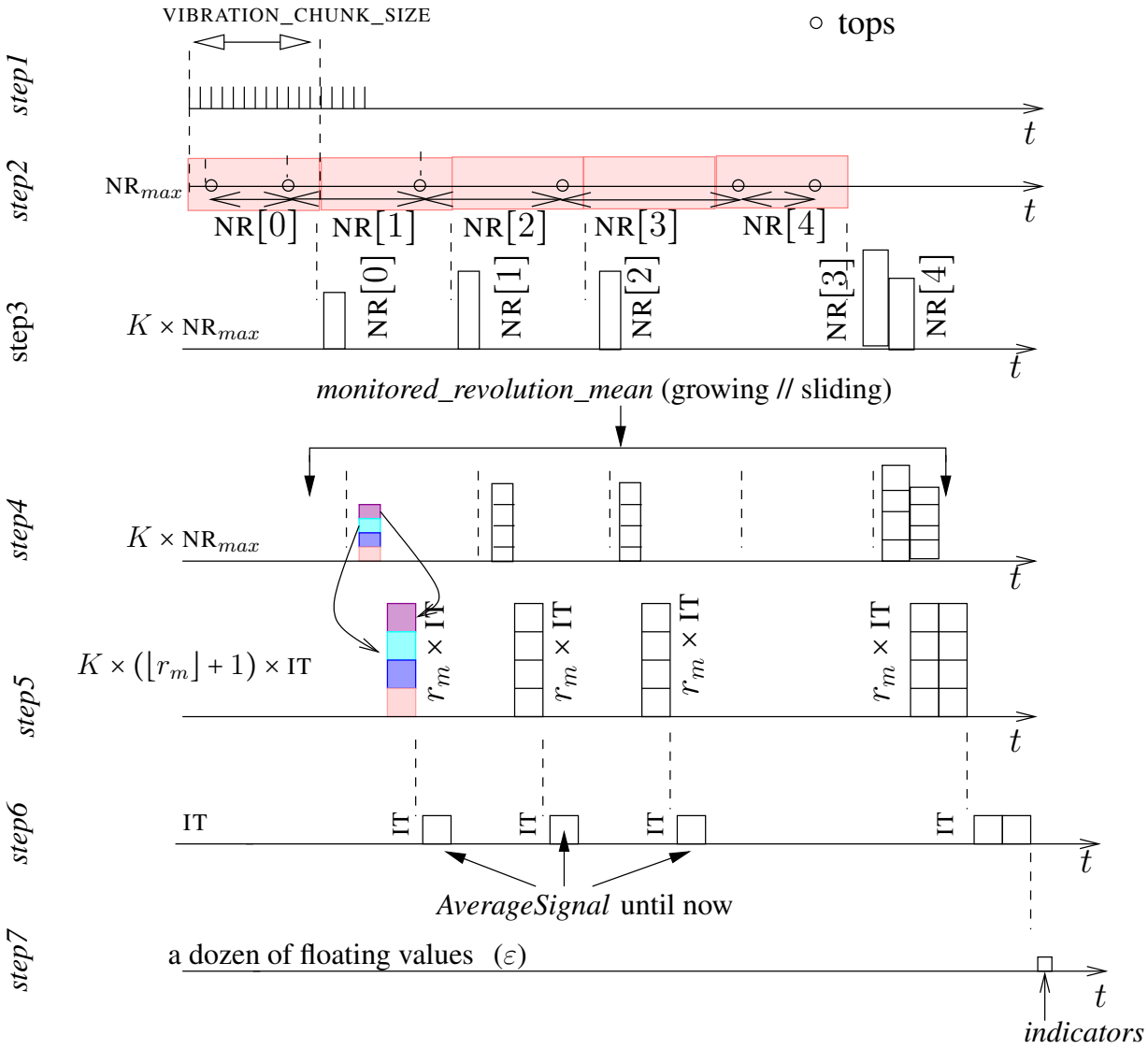


Figure 5.12: Workload Estimation

5.11 Conclusion

We have transformed an on-ground reference computation of a health indicators, in which the global average *monitored_revolution_mean* is used to estimate the positions of the tops of the monitored shafts, into an incremental embedded real-time computation of the same indicators, in which the average is calculated using either a growing or a sliding window.

We have first compared the growing indicators with the reference ones and then the sliding indicators with the reference ones. These experiments are performed on numerous acquisition files and have shown that there indeed exists a difference. In addition, there is no way, based on our implementations, to decide that this difference is acceptable or not. However, what really matter are the maintenance decisions taken by the operator. He/She raises the maintenance decisions based on the trend of the indicators, not on absolute values. The reference HMS defines the trend as a moving average of the indicators. For a given vector of indicators (IND(N)), the trend vector is calculated in order to smooth the evolution of the

indicators as follows:

$$\text{TREND}(i) = \frac{\sum_{j=i-5}^{i+4} \text{IND}(j)}{10}$$

It means that, the trend is calculated by averaging 5 indicators before and 4 values after. In all our experiments, given the same input files, each incremental implementation has the same trend as the reference one therefore this will lead to the same maintenance decisions. We will also present the results of the new incremental algorithms (not necessarily running on the embedded platform) to the experts in charge of the maintenance decisions. As mentioned previously, their decision should be the same. Some experts have been already asked to look at the results and they confirm that the difference is acceptable, but the complete procedure has not been conducted in full for now.

Moreover, we have performed many experiments on various monitored shafts to compare the growing and sliding window principles. These experiments have shown that the indicators calculated using a growing window are closer to the reference ones. Our embedded platform has a limited storage capability (2MB) in each cluster (see Chapter 7). The workload estimation allows us to estimate the amount of memory required to go from vibration samples to functional data used to compute indicators. This estimation is very fine-grain and considers the sampling frequency, the speedup ratio, the interpolation size, the variation speed of the reference rotor and the acquisition range.

For the future, if the idea of computing such algorithms on an embedded platform is adopted, it means that the very first design of the algorithms has to take this constraint into account: global averages, for instance, will be forbidden. The complete approach we followed can be used as guidelines for the transformation of other signal processing applications, which are likely to exhibit the same kind of algorithms.

Part II

Implementation on the manycore MPPA-256

Chapter 6

Choosing a Many-core Processor

6.1 The advent of many-core processors

An empirical law deduced from a simple observation has been for many years the roadmap of the microprocessor industries. This law was first announced in the *Electronics* magazine by *Gordon Moore*, one of the co-founders of *Intel*. This law states that the number of transistors on a silicon chip doubles every two years. For several years this prevision was right. Between 1971 and 2001, the density of transistors on the market doubled every 1.96 years. This technological evolution allowed the general public to access to computer equipments. Then, it was possible to have a chip twice more powerful with the same price.

This law has encountered limiting factors:

- The first disadvantage to fulfill the Moore law is of physical nature. Today, the most advanced integrated circuits are engraved with a fineness of 14 or 16 nanometers. According to [32], manufactures intend to descend first to 10, then to 7 and even 5 nanometers. At 2-3 nanometers, we reach the limits such that the behavior of the transistor no longer obeys to the rules of physics, but to the rules of quantum physics.

- The second drawback about Moore's law is related to heat dissipation. The integration of more and more transistors in the same integrated circuit generates heat dissipation.

- In addition, another limiting factor is financial aspect. In fact designing thinner and thinner transistors increases the production costs of integrated circuits such that IBM and Siemens joined their investments to follow the trend ([33]).

- The performance of a mono-processor could be upgraded by improving the microarchitecture of the processor. However, doubling the complexity of a processor allows to gain *40%* of performance according to [34] Pollak in 1999. Complexity refers in this context to processor logic, for instance *arithmetic logic unit (ALU)*.

To solve these problems, the solution that seemed the most obvious was to increase the performance thanks to a parallel architecture. The first multi-core processor (*POWER4*) was designed by *IBM* in 2001.

Basically, each core of a multi-core processor is a processor with a single core, except that L2 and L3 caches are often shared to enable all the cores to access it. Multi-core processors were mainly introduced for two main reasons: *power consumption* and *scalable performance*.

- **Power consumption management:**

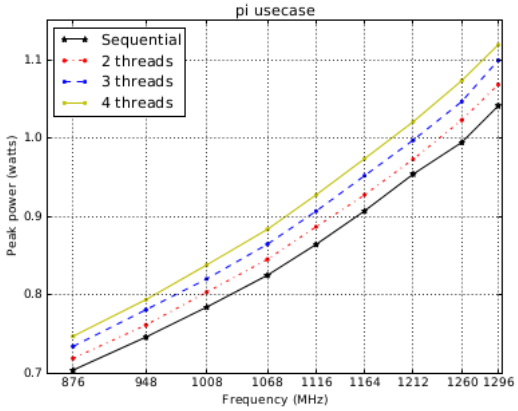


Figure 6.1: Peak power of the pi use case in all configurations

Designers have turned into multi-core processors to make powerful platforms while ensuring stable power consumption. For example, Paolillo ([35]) describes the impact of power consumption when switching from a single-core processor to a multi-core one. According to [36], the instantaneous power of a multi-core processor with homogeneous frequency and voltage is given by:

$$P(f, V_{dd}, k) = k(\alpha V_{dd}^2 \times f + \beta V_{dd}) + \gamma$$

where f is the operating frequency, V_{dd} is the supplied voltage and k is the number of active cores. α, β and γ are inherent constants depending on the processor manufacturing technologies, such as switching capacitance.

This equation states that the power consumption linearly increases with the number of computing units (cores). Whereas voltage and frequency can cause a quadratic or cubic increase of the power consumption. This intuition was confirmed by the experimental results described in [35]. The experiment is performed on multi-core (4 cores) based on ARM Cortex A9. The example is developed in C with the OPenMP library to tackle threads parallelism. This experiment consists in measuring for a given frequency of the processor the power consumption. The power consumption is measured in four scenarios: one core (sequential implementation), 2 cores, 3 cores and finally 4 cores. Each thread computes an approximation of π by using the *Riemann* method. This experiment is chosen because it is easily parallelizable and the computation load can easily be distributed over cores.

Figure 6.1 shows experimental results: it is more advantageous in terms of dissipation to increase the number of cores rather than to increase the frequency. *For instance, a single core running at* 1008MHz *dissipates more than 4 cores, each operating at* 876MHz.

Today, in multi-core processor domain a common technique is usually used to facilitate energy management. This technique consists in turning off unused cores and dynamically decreasing the voltage and the frequency when the processor load is low (DVFS).

- **Scalable performance:**

  Multi-core processors are used in many applications, especially for applications where high performance is required.

  For example, multi-core processors are used in the field of computational fluid dynamics to simulate the movements of fluids. *Navier-Stockes* equations are used to model the movement of these

fluids. These equations are partial differential equations. The finite element method is used to solve these equations. This method consists in splitting the spatial domain of the problem into small finite elements called mesh and resolve equations in each mesh. This type of computation is well suited to parallel computation because the computation of each mesh can be parallelized. According to [37], Sakharnykh simulates incompressible, viscid fluid flows on a multi-core processor and obtained an about 9.5x speedup compared to a single CPU core implementation.

Multi-core and Many-core processors have also revolutionized weather forecasting. Weather prevision is an area that requires considerable computation capabilities. To make a long-term forecast, it is necessary to take into account all the globe and split it into several meshes: the globe is first divided into longitude and then in latitude. According to [38], the simulation of the evolution of the climate system (atmosphere, ocean) has become faster in the last twenty years. Twenty years ago, 3 months of computation were needed to simulate climate change over 100 years with a simple model. Today, 10 days of computation are enough.

6.2 Challenges for programmers and vendors

However, the performance gain provided by the many-core processors comes with several challenges for both programmers and vendors. A many-core processor does not necessarily mean that the application we are developing will run faster. Most beginners in parallel programming are enthusiastic about having a powerful platform and are excited to write their first parallel program. They realize that their first program does not have the expected performance and sometimes gives worse performance than the same program executed on a single-core processor. We need a deep understanding about how software and hardware are working to build an efficient parallel application.

6.2.1 Software point of view

Programmers must now design applications with multiple threads so that it is possible to parallelize their processing. They should be able to identify when their applications can be divided into several independent tasks that can be executed at the same time on different cores. Sometimes the transition from a sequential application to a parallel application can be very arduous and require to rethink all the design. Programmers face many challenges to deliver applications whose performance scales with the number of processing units. The main challenges are the following:

1. *Communication*: in a multi-threaded application, usually each thread performs its processing in parallel and at the end its results are read by another thread. The communication time between threads in a parallel application is very important. Depending on this communication time, the performance of the application may be chaotic. Adding one core can drastically reduce performance. Moreover, the programmer must choose between *synchronous* or *asynchronous* communications. In synchronous communications, the cores wait at determined points for exchanging data. In asynchronous communications the cores no longer have constraints on specific points to exchange data. But the programmer must ensure data validity. For example, when a core reads data from another core, it will be necessary to ensure that data is updated, or not overwritten in memory.

2. *Load balancing*: Programmers must divide their programs in such a way that workload is fairly balanced between cores.

   A load balancing issue may lead to a situation where cores with low computational load wait cores with more computational load. This unbalance of charge leads to an under-utilization of the resources whereas the ideal would be to keep the cores busy as much as possible.

3. *Data consistency*: The data consistency is fundamental in a multi-core architecture to prevent the cores from computing algorithms with erroneous data. The difficulty of managing data consistency depends on the type of architecture: *distributed memory* or *shared memory* architecture. In the case of a distributed memory, each core has its local memory. The core is the only one that can read and modify local memory. On the other hand, data consistency is more difficult to manage in a shared memory architecture. For this type of architecture, each core has a local cache and can access a shared memory. The programmers should ensure that the cache and shared memory data are consistent all the time.

6.2.2 From the vendors point of view

Manufacturers must provide developers with tools and frameworks to facilitate the deployment on their many-core architectures. These tools should help develop applications very quickly and effectively exploit the resources of the multi-core processor. The best use of hardware resources requires some expertise. That's why some manufacturers like Intel provide a set of tools for their *Xeon processors*. These tools are libraries with predefined functions that are suitable to the platform. These functions are easily added to the source code. Intel provides also a compiler for their processors.

6.3 General overview of a many-core processor

Many-core processors have emerged during the last decade, as an evolution of *multi-core* processors. In multi-core processors, a relatively small number of *cores* called also *PEs* (Processing Elements) are connected on chip through a bus, sharing the same memory. Many-core processors are usually structured into *two layers*: cores are grouped into *clusters*, in which they may share a memory with a bus, like in multi-core processors. Several clusters are connected through a *network-on-chip (NoC)*.

6.3.1 A many-core processor

A many-core processor is characterized by:

- *an important number of simpler cores*: the number of cores can go from few hundreds to few thousands. According to [39], the fastest computer in the world is the *TaihuLight* in 2017. This computer is composed of 40,960 SW26010 *many-core* processors with a performance of 93.01 PFLOPS.

- *a new interconnect*: the historical way that the cores in the same processor communicate in a shared memory is to use a *bus*. However, the bandwidth and the latency become a major issue when the number of cores increases. Then, according to [40], new interconnects have emerged like *ring buses*. They also face a scalability issue when the number of cores increases. Consequently, the Network on Chip (NoC) brings significant improvements such as *throughput* and *scalability* in comparison with classical interconnects such as buses, *ring buses* and *crossbars*.

- *new memory architectures* which offer higher memory bandwidth access using independent memory banks.

6.3.2 Sources of Interferences in a many-core processor

The timing interference is defined in [41] as the interference delays due to conflicting accesses to concurrent hardware resources (cache, bus, memory, network etc.). For instance, Figure 6.2 illustrates the

difference of memory timing accesses depending on the type of platform (single core and multi-core). In this example, two tasks are running concurrently. In a single core, the memory timing access mainly depends on the memory access region. However, in a multi-core, conflicting accesses increase the overall execution duration of the tasks. Indeed, the memory bus arbiter operates according to a given policy. The bus arbiter can serve only one task at a given time and the others have to wait, leading to an extra delay.

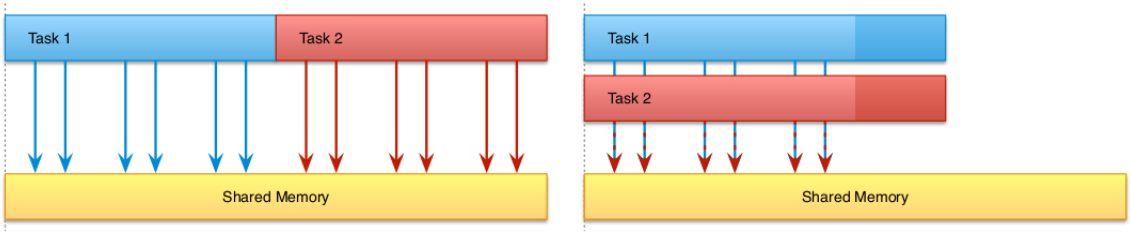


Figure 6.2: Two tasks are accessing a shared memory on a single core (left) and a multi-core (right). Figure is extracted from [41]

.

Traditional many-core interferences have different sources: (i) inside a cluster due to shared resources (memory, bus), (ii) interferences between clusters on the NoC, (iii) interference on the I/O cluster to access the SDRAM.

Inter-cluster interferences. Clusters exchange data using the NoC. The NoC is composed of a set of routes and routers. Each cluster has one router which can redirect packet to other routers. The NoC traffic is regulated by a routing policy, for instance a wormhole switching. The NoC is a source of congestion and could have a great impact on the performance of the application. According to [42], some applications are more *sensitive* to the NoC interferences than others. These applications require a high traffic over the NoC compared to *memory intensive* applications.

Intra-cluster interferences. A cluster has the same configuration as a classical multi-core processor. Therefore, a cluster addresses the same interferences. According to [43], a classical multi-core has the following interferences: (i) interference on cores due to preemption, (ii) interference on private cache due to hardware coherency policy and (iii) shared memory and bus interferences.

SDRAM interferences. Each computing cluster can access the external SDRAM memory through the I/O clusters. A read operation invoked by a core (more precisely a thread) of a computing cluster crosses the NoC and the I/O cluster and finally reaches the SDRAM controller. According to [43], the SDRAM interferences is mainly caused by refresh cycles.

6.4 Examples of Many-core Processors

6.4.1 Tilera TILEPro64

Overview

TILEPro64 ([44]) belongs to the general-purpose class of many-core processors. According to ([45]), the TILEPro64 processor supports a wide range of intensive applications including advanced networking, digital multimedia and wireless infrastructure. It is the second generation of many-core processors from Tilera. As depicted in Figure 6.3, TILEPro64 processor is composed of 64 general purpose cores organized in 8×8 two dimensional array.

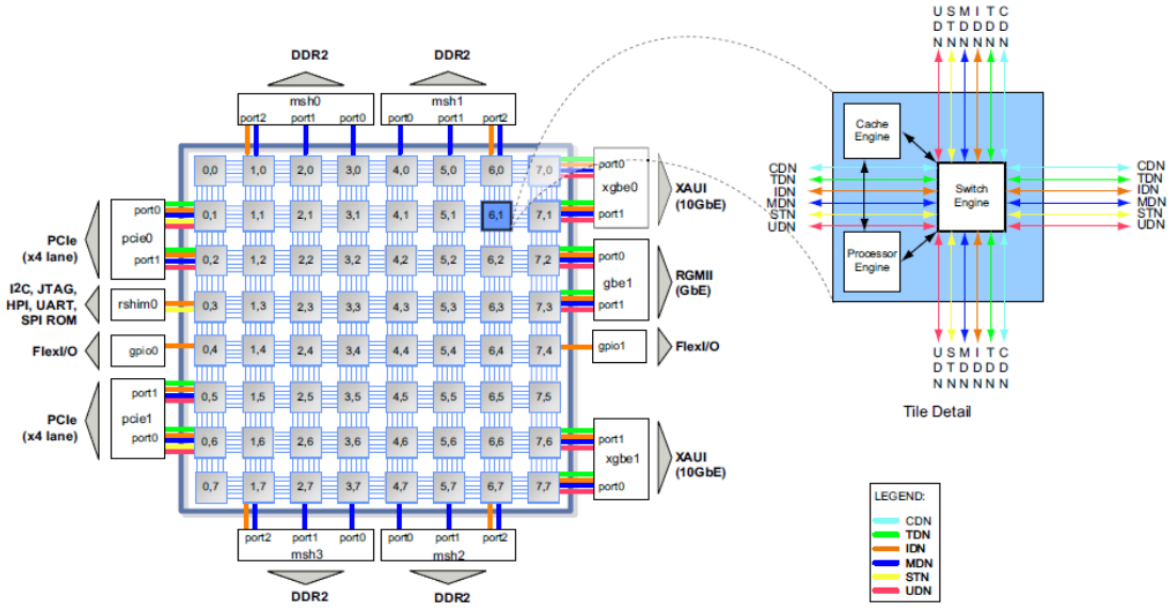


Figure 6.3: High-level overview of the TILEPro64 many-core processor architecture.The processor is an 8x8 two dimensional array. Each tile has a VLIW CPU, a total of 88KB of cache, MMU and switch engine. I/O devices and memory controllers connect around the edge of the mesh network. The figure is extracted from ([46]).

Tile

As illustrated in Figure 6.3, each tile is composed of a core engine, cache engine and switch engine. The core engine is a VLIW processor with 64 registers. Each core has also L1 and L2 caches. The L1 data and instruction cache size is respectively 8 and 16 KB. Finally, the 64 KB L2 cache combine data and instructions. This provides a total of 5.5MB of on chip cache for the TILEPro64. The L2 caches are accessible by any core and I/O devices. The cache coherency is managed by two dedicated networks. Finally, each tile contains two DMAs engines responsible for orchestrating memory data streaming between tiles and external memory, and among the tiles.

NoC

The different tiles are connected through a Network on Chip (NoC). Six independent networks interconnect and connect the tiles to the I/O devices. Three NoCs are only used for hardware devices and cannot be accessed by the user. They manage the data transfer and cache coherency between cores. Manel in [46] has detailed the goal of each of these NoCs:

- The Memory Dynamic Network (MDN) is used for memory data transfers between the tiles and the 4 memory controllers of the chip. Only the Cache Engine has a direct hardware connection to the MDN.

- The Tile Dynamic Network (TDN) is also dedicated to memory traffic. It is used for transferring data between the caches of the tiles. Again, only the Cache Engine has a direct hardware connection to the TDN.

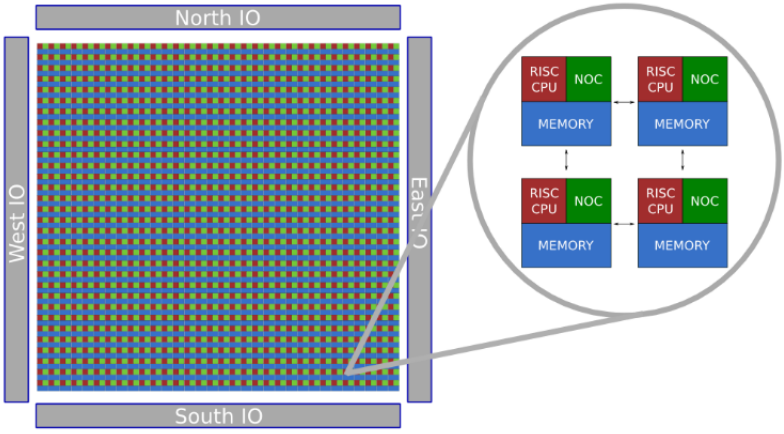


Figure 6.4: High-level overview of the Epiphany-V many-core processor architecture. The processor is an array of 1024 cores. Each tile has a 64-bit RISC CPU, 64KB of local memory and a NoC interface. The figure is extracted from ([47]).

- The Coherence Dynamic Network (CDN) is also dedicated to memory traffic. It only carries cache-coherence invalidate messages between the tiles.

The other three networks are allocated to the software. One is used to perform data transfer between I/O and tiles, and I/O and memory. The remaining two others are used for low latency data transfer between tiles.

6.4.2 Adapteva Epiphany

Overview

The first Epiphany many-core processor contains 16 cores and was released in May 2011. The second generation gathered 64 cores arranged in a 8×8 2D array meshes. In fact, the latest *Epiphany-V* many-core processor ([47]) depicted in Figure 6.4 is composed of 1024 cores, 1024 programmable I/O pins and has a total of 64MB on chip memory. Epiphany-V processor is designed to address energy efficiency, peak performance limitations in real-time communication and image processing applications. This processor supports floating point operations and achieves 50 GFLOPS/W in 28 nm technology.

Tile

Each tile contains 4 memory banks for a total of 64KB. The memory architecture does not have traditional L1/L2 caches and is organized in a distributed memory system where each tile can access the others tiles. According to [48], the scratchpad SRAMs have much higher density than L1/L2 caches since they don't include complicated tag matching and comparison circuits.

The core of the Epiphany-V processor is named in the literature *eCore*. It is a RISC architecture that supports the IEEE754 floating-point unit (FPU), an integer arithmetic logic unit (ALU) and a 64-word register file ([48]).

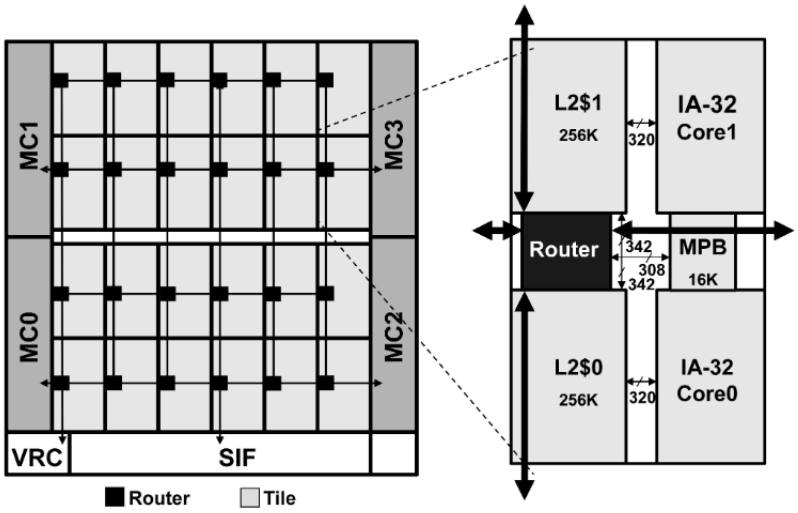


Figure 6.5: High-level overview of the Intel SCC many-core processor architecture. The processor is an array of 6x4 tiles. Each tile has two cores and a router interface. The figure is extracted from ([50]).

NoC

The different tiles of the Epiphany-V processor are interconnected with the chip through three independent 136-bit wide mesh networks. These three channels are labeled: *rMesh, cMesh* and *xMesh*. According to [46], the *cMesh* and *rMesh* NoCs used a classical 2D mesh topology and are based on a static X-first or Y-first routing algorithm and fair (round robin) arbitration. The *xMesh* NoC only allows a packet to travel in one direction (up, down, left, or right) without changing direction.

These three networks have different purposes:

- *cMesh*: has a high bandwidth and is used for on-chip write transactions between tiles. It connects the node to all four of its neighbors.
- *rMesh*: used for all read requests, read between tiles or between one tile and the exterior of the Epiphany-V processor. The *rMesh* network connects also the tile to all of its four neighbors.
- *xMesh*: used for write destinations involving the off-chip resources.

6.4.3 Intel SCC

Overview

SCC is a many-core processor designed by *Intel* ([49]). Its name is inspired by the concept of *Cloud Computing* and belongs to the general purpose class of many-core processors.

The *SCC* many-core processor in Figure 6.5 integrated 48 cores organized in two levels of parallelism. The first level of parallelism is composed of 24 clusters (matrix 4×6). The second level is the two cores in each cluster running in parallel. Each core can boot on a separate Operation System (OS) and behaves like an independent PC. According to [46], SCC routers on the NoC are each equipped with 8 virtual channels. They allow to forward packets traversing the NoC using different priorities. But the user has no control over the definition of these priorities. The SCC processor brings innovation in terms of energy efficiency and techniques that allow to attain power consumption from 125W as low as 25W.

Tile

Each tile is a cluster of two cores with IA-32 instructions sharing a router for the tile communication. According to [50], the separate L1 instruction and data caches are upsized to 16KB and support both write-through and write-back. Each L1 cache is reinforced by a unified 256KB 4-way write-back L2 cache. The L2 caches correspond to a total of 12 MB on-die. A Message Passing Buffer (MPB) is also present in every tile and is dedicated for shared-memory communication between tiles. There is no hardware cache coherency for L1 and L2 caches, the memory coherency is managed at software level using for instance OPENMP ([51]) or MPI ([52]).

NoC

The 24 tiles are connected on the NoC through a 2D torus topology with bidirectional links between routers. The X-first protocol is used to route data on the NoC. According to [46], the NoC routers feature 8 virtual channels used to implement 8 priority levels among packets traversing the NoC. However, these priority levels are not user-defined.

6.4.4 MPPA-256

The many-core processor MPPA-256 is printed in details in Chapter 7, page 97. Here, we will present a general overview of the processor.

Some recent developments in the microprocessor industry have designed many-core processor to address the embedded systems market. This is the case of the *Kalray MPPA-256 (Multi-Purpose Processing Array)*. The global architecture of the *MPPA-256* is depicted in Figure 6.6. It integrates 16 *Computing clusters* and 4 *I/O clusters*. The I/O clusters are represented on the four sides of Figure 6.6. Each I/O cluster has four cores. Clusters are connected by a NoC (Network on Chip). The overall architecture integrates 288 cores (each computing cluster and I/O cluster has respectively 16+1 and 4 cores : $16 \times (16 + 1) + 4 \times 4$).

Tile

Each tile is composed of 16 computing cores and 1 DMA that manages the data transmission between the tiles through the NoC. The shared memory can be configured in memory bank mode. In this configuration, each core has a dedicated memory bank and bus arbiter that will reduce interferences. In addition, the cache is managed at software level by using data cache primitives.

NoC

The NoC is designed with 2D links: data and control transmission between the tiles. The user can configure the NoC. It is a source of congestion but the worst case traversal time is guaranteed.

6.5 Conclusion

In many-core processors, the potential interferences are bad for predictability and response-time guarantees. Not all many-core architectures are suitable for avionic systems. Some processors are designed to achieve high average performance, but offer no guarantees on individual executions. In particular, the sources of non-predictability of timing are numerous: they can be due to the complexity of the cores themselves, or to the access to shared resources like buses, networks-on-chip (NoCs) and the memory.

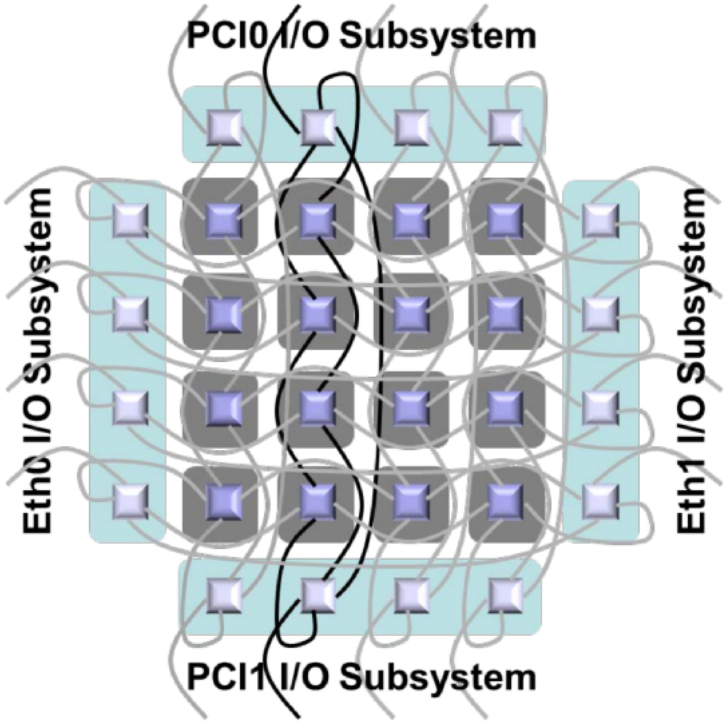


Figure 6.6: Abstract view of the MPPA-256 Many-core Architecture (extracted from [53])

The MPPA-256 many-core processor reduces the number of contention points (they do not remove all of them). Each core is a simple VLIW architecture, which allows predictability. In particular, the dynamic optimizations present in a lot of off-the-shelf microprocessors, like branch prediction and out-of-order execution are forbidden. Optimizations are pushed back into the compiler, which brings hardware predictability. As a counterpart, it lowers the source to object code traceability which is usually used to achieve some DO-178 objectives. According to [54], the benefits of the Kalray platform for critical real-time systems are: deterministic computation, deterministic and predictable response times, low power and high performance.

Chapter 7

Towards using MPPA-256 Processor in Avionic Industry

7.1 Avionic System Requirements

The term *avionic system* encompasses both the on-board hardware and software which perform various functions such as the fuel level management, the navigation (automatic pilot), the communication (message exchanges with the ground) etc.

Usually, these systems continuously interact with the environment at a speed determined by the environment. They are required to correctly function because an error can lead to tragedies. To prevent disasters, certification authorities have defined certification rules for avionics software for example *DO-178* according to [5].

In addition, they have to respect their *timing constraints*. In fact, the timing constraints are defined by the *latency* requirements between sensors and actuators or between sensors and display units (human factor). We need to guarantee the timing correctness of the system, for instance missing deadlines will never occur. In order to ensure that applications satisfy their deadlines, an upper bound of the execution time is required (WCET).

According to [55], there are two families of approaches to perform a WCET analysis: *static WCET analysis* and *measurement based-WCET analysis*. The static approach considers all possible input values of the application but it is not executed on a real hardware. Its accuracy depends on the timing model of the hardware. However, the measurement based-WCET method is executed on a hardware or a simulator. Only representative input values are tested. But this approach suffers from no guarantee that the tests cover the worst case execution time.

According to [55], combining these two methods provide additional assurance of the correctness and precision of the approaches on a simple processor with a single core, no cache, no speculative features etc., it is possible to tightly bound the worst-case execution times (WCET) either via static analysis or via measurements of all relevant paths as illustrated in [43]. However, the shift from single core to more complex processors (multi/many-core processor) raises new issues for both approaches due to multiple contention for multiple shared resources between co-running applications in the same processor. The WCET analysis should model the communication through the NoC (Network on Chip), determine the behavior of memory hierarchy access.

7.2 Architectural Improvements: from federated architectures to IMA

Few decades ago, each avionic function was managed by a single computer (*one function = one computer*). For example, a computer is dedicated to the automatic pilot function, another one to the flight management function, etc. Each of these functions uses its dedicated resources, sensors and actuators. This concept is known in the literature as *federated architectures*.

According to [56], the federated architectures have many drawbacks: software changes on one module will affect other modules because modules are connected through point-to-point wiring; the response time of such systems is very slow and difficult to upgrade. On the pro side, these systems are robust and can function correctly despite failures at others points in the system.

However, since the 90s, airlines want more and more sophisticated functionality such as *entertainment systems for passengers*. Therefore, this concept reaches its limitation for two main reasons. The first one is related to the weight and power consumption. The second reason is related to the maintenance cost with this growing number of computers.

7.2.1 IMA principle

IMA (Integrated Modular Avionic) was then proposed to allow current computers architectures to manage more than one function. The IMA concept permits to reduce the number, weight and cost of computers. It is first implemented on *Boeing 777* Airplane Information Management System (AIMS). According to [1], the IMA approach is based on two standards: the *ARINC 653* [57] which defines the partitioning (resources and temporal) and the *ARINC 664* [58] which defines the principle of the communication between several functions.

An example of IMA architecture is described in Figure 7.1. It is composed of a set of *Computing Processing Modules (CPM)*. In this example, we consider 3 Computing Processing Modules: CPM1, CPM2 and CPM3. These modules are connected by switches to exchange data thanks to the *ARINC 664* standard. However, each module must respect its communication constraint in order to avoid a communication bottleneck and more important to make the communication predictable. A communication constraint may be for instance a bandwidth limitation for each virtual link. According to [1], thanks to the communication constraints, it is possible to formally prove that: no message is lost during the communication and the end-to-end communication of a given message is bounded.

The IMA principle supports the resources partitioning which allows the co-existence of several applications sharing the same platform. For instance, in Figure 7.1, the partitions A1, A2 are running on CPM1; B1, B2 and B3 in CPM2 and C1, C3 are hosted in CPM3. Moreover, these partitions sharing the same CPM are partitioned with respect to the space (resources partitioning) and the time (timing partitioning).

1. *Resources partitioning*: Each partition has its dedicated resources like memory, I/O resources such as the bus ARINC 429 and AFDX (refer to [59]) etc. This allocation is performed in a static manner. It is up to the system integrator to allocate enough resources to each partition wrt. usage domain and to ensure their segregation.

2. *Temporal partitioning*: The system integrator defines the scheduling of different partitions running on the same CPM by using configuration files. This scheduling is defined statically. Each function has a dedicated time slot. For instance, in the CPM1, during each period, the partition A1 is executed first during its time slot. Then A1 is suspended and the execution is given to A2. This sequence is repeated periodically.

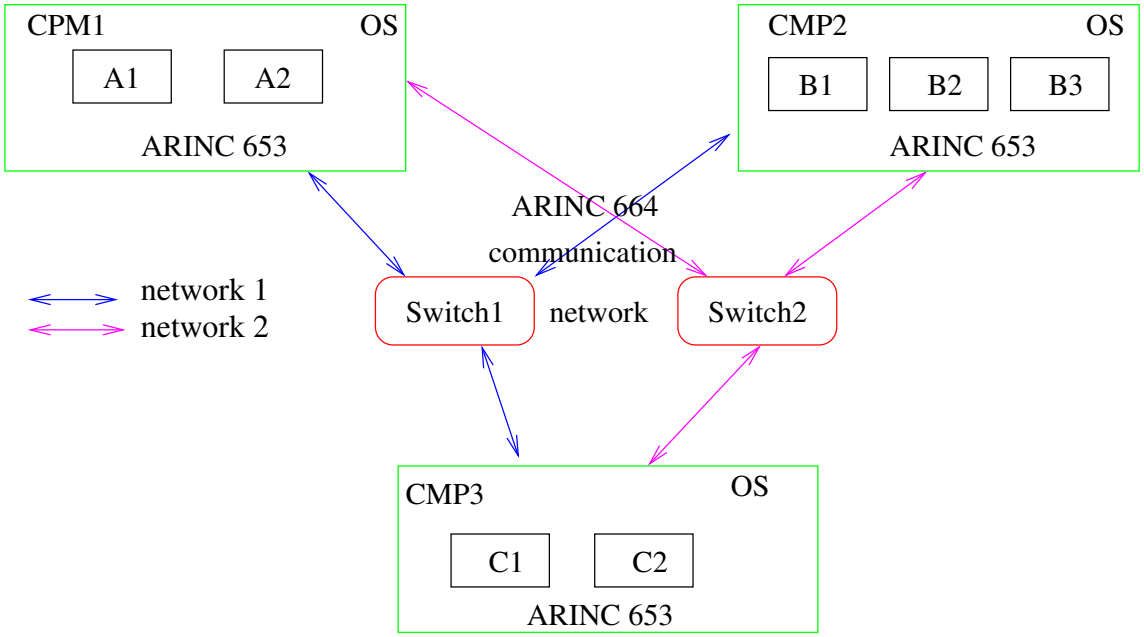


Figure 7.1: Typical IMA architecture (inspired by [1])

7.2.2 SWaP: Size, Weight and Power

The SWaP (Size, Weight and Power) reduction is an important concern for avionic architectures. According to [60], with the increasing number of sub-systems to inter-connect, the wiring length and weight increase and sometimes reach 100s of kilometers. In addition, the research presented in [60] has performed a comparative study in terms of size, weight and power between the federated and IMA architectures. This quantitative study is done using data of twenty-eight different projects.

Size reduction. The volume reduction is examined using eleven projects as illustrated in Table 7.1.

Aircraft	**Volume reduction in %**
Boeing 777ER AIMS	50
Airbus A380	50
Rafale	50
Cessna Citation	40
Challenger 601	28.28
Falcon 20	28.28
Falcon 50	28.28
King Air 350	50
King Air B200GT	50
King Air C90GTi	50
King Air 300	28.28

Table 7.1: Percentage volume reduction (extracted from [60])

This table shows that the switch from federated to IMA architectures comes with a gain of size reduction that ranges from **28.28%** to **50%**. This reduction is much more significant for large-size aircraft such as Boeing 777ER and reaches 50%.

Weight reduction. In the same way, the weight reduction is also evaluated from different projects. The weight reduction allows a payload improvement. The observed weight reduction when replacing the federated architectures to IMA ones is between **25%** and **50%** according to [60]. The weight shedding is 50% for aircraft like *King Air 350*, *Boeing 777ER* and *King Air B200GT*.

Power reduction. The transition from federated architectures resulted also in a diminution of the power consumption. The power reduction is **38%** for aircraft like *Challenger 601, Falcon 20, Falcon 50, King Air 350* and a maximum reduction of **60%**.

7.3 MPPA-256 many-core processor

Real-time functions state that deadlines are as important as functional requirements. Therefore, worst case execution time verification is as important as functional requirement verification. This constraint is taken into account in the IMA architecture by resources, timing partitions and the bandwidth limitation on virtual links. It is also addressed in the many-core architecture by choosing a processor that offers determinism and predictability mechanism since the design.

However, resource sharing makes the estimation of the worst case execution time very challenging because we can estimate all the interferences but it leads to very pessimistic WCET, in particular when the memory coherency generates implicit communications. The MPPA-256 processor offers mechanisms to remove interferences (not all) between cores since the design.

7.3.1 Computing Clusters

A computing cluster is illustrated in figure 7.3. It is composed of 16 computing cores (each core is called a *Processing Element* or *PE*) plus one system core (Resource Management). Each computing cluster has also one DSU (Debug System Unit). Every DSU has a 64 bit counter. This counter is mapped at a specific adress in the memory and can be read to get its current value (will be discussed in Section 8.1.3, page 109). In addition, a cluster has also 2 DMAs (Direct Memory Access) to exchange data; one for transmitting data and another one for receiving data. They concurrently access the 2MB shared memory.

Inside each cluster, the local memory is composed of 16 independent memory banks. Each memory bank is composed of 16384×64-bit words. A memory bank can be accessed via a dedicated arbiter which reduces the interferences between cores compared to a single bus arbiter. This memory can be configured in two different ways: *interleaved mode* (Figure 7.2a) and *blocked mode* (Figure 7.2b). In interleaved mode, words with consecutive addresses are stored in consecutive banks. More precisely, sequential addresses move from one bank to another every 64 bytes as illustrated in Figure 7.2a. In contrast, in a blocked mode, each block of 128 kB consecutive memory addresses is contained in the same memory bank.

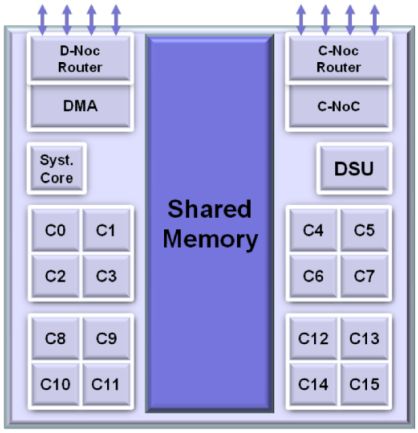
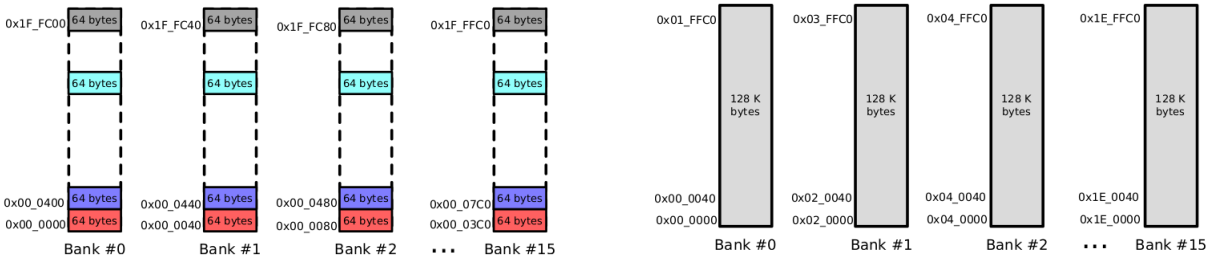


Figure 7.3: MPPA-256 Compute Cluster (extracted from [53])



(a) Interleaved address mode

(b) Blocked address mode

Figure 7.2: Interleaved and Blocked address mode configuration of the 2MB shared memory (extracted from [54])

According to [54], the blocked address mode is better suited for time critical applications to reduce interferences because access from different cores go through different bus arbiters.

7.3.2 I/O (Input/Output) Clusters

The four I/O clusters are represented on the four sides as illustrated in Figure 6.6. Two of them are connected to a PCIe controller and two others to an Ethernet controller. Each IO cluster has 4 cores and 4GB DDR- SDRAM memory. In addition, IO cluster manages also the link between the HOST and the computing clusters.

7.3.3 Network on Chip (NoC)

Clusters are connected to each others by 2 links. The first link is dedicated to the data transmission (D-NoC) and the second one is about control data (C-NoC). In our case study, the D-NoC is used to transmit the raw vibration data while the C-NoC allows to synchronize data transmission between clusters. The user can explicitly configure in static manner the route of the transmitted data by indicating the clusters source and the destination. According to [61], the D-NoC is based on wormhole switching which offers guaranteed bounds on Worst-Case Traversal Time (WCTT) in specific configurations such as no deadlock route, flow limitation by the source node.

7.3.4 MPPA-256 Interferences

We have identified *three* sources of interferences in the MPPA-256: (i) cores in a cluster access concurrently the 2 MB shared-memory; (ii) NoC is a source of congestion: one data arriving from 2 links has to be sent to the third one; (iii) Interferences in the external Synchronous Dynamic Random Access Memory (SDRAM) memory in the I/O cluster are mainly due to refresh cycles. The software developer cannot avoid the refresh and when it occurs in time.

The MPPA-256 provides hardware mechanisms to reduce the two first categories of interferences: memory banks configuration and NoC bandwidth management which guarantees a precise worst case traversal time (WCTT).

7.3.5 Data Transmission between Clusters

Clusters exchange data through the NoC using special communication objects such as *portal* and a *channel*. The communication can be *synchronous* or *asynchronous*. In synchronous communication, the sender blocks as long as the message is not stored in the destination space (resp. the receiver blocks as long as the expected data is not stored in the reception space). On the contrary, asynchronous communication means that the message is put in a queue and the control flow returns immediately.

A *channel connector* allows a synchronous point to point communication. The reading function is blocking until data is available. In the same way, the writing is also blocking until the reader receives the transmitted data. When two clusters are exchanging data, each one needs to open two channels, one for sending data and another one for receiving data.

A *portal connector* involves N writers and 1 reader (N:1). It performs an asynchronous communication. The writer puts data in the reader shared memory without the reader being involved.

7.3.6 Synchronization mechanism between Clusters

To ensure the integrity of the transmitted data, MPPA-256 IPC (Inter Process Communication) offers synchronization mechanisms, for instance a *sync* connector. The sync connector (N:1) involves N writers and one reader. It allows to synchronize N writers with one reader. The reader sets an initial value (a flag) and performs a blocking read operation. Each sender sends a value which is *OR-ed* with the current flag on the reader side. When the OR-ed value equals -1 (all bits of the flag are 1) then the reader is released.

7.3.7 Data Transmission between cores

All the cores in a cluster share the same memory: they can read (resp. write) data from (resp. in) the memory. The term *shared memory* means that all the cores have a common address space. Suppose that core1 reads the data X which is located at the address 100 and moves it into a CPU register. If the core2 performs the same read operation, they both are referring to the same physical address 100.

In addition, each core has a separate data and instruction cache of 8KB each. The cache coherency is managed by the developer at software level by using data cache primitives. For instance, a full memory barrier in *write-through mode* is implemented as follows:

$\_\_builtin\_k1\_wpurge()$; /* request write buffer flush to memory */
$\_\_builtin\_k1\_dinval()$; /* invalidate the whole data cache */
$\_\_builtin\_k1\_fence()$; /* wait for all memory operations to commit */

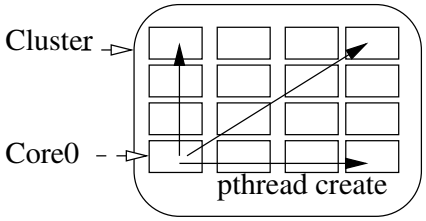


Figure 7.4: Create Threads on one MPPA Cluster

7.3.8 Synchronization between cores (threads)

Synchronization mechanisms play a crucial role in shared memory programming. They allow to control the access to shared variables between threads. According to [62], the synchronization between threads can be implemented using either software locking algorithms like *mutex, semaphore* or *hardware atomic primitives*. On one hand, mutex and semaphore are blocking and not scalable. On the other hand, atomic functions allow to update data without blocking when data does not exceed the size of the bus.

7.3.9 POSIX Programming Model on MPPA-256

Threads programming was invented as an abstraction level to explore parallelism. According to [63], historically, each hardware manufacturer has implemented its own version of threads that makes it hard for programmers to develop portable applications. POSIX is an acronym for Portable Operating System Interface. According to [64], POSIX is a family of standards specified by the IEEE Computer Society for maintaining compatibility between operating systems. According to [63], one of the advantage of using threads instead of unix process is that a thread is created and scheduled with less operating system overhead. In addition, a multi-threaded application also offers performance gain compared to a mono-threaded application. For instance, threads allow to overlap the computation and the I/O communication. While one thread is waiting for new arrival data, others threads can perform intensive computations. The main services used in POSIX programming are: threads creation, signals, timers, threads scheduling, threads synchronization.

The gain of performance offered by thread programming comes with several challenges. The programmer must be able to divide and organize its application into many independent threads. According to [63], there exist three models for threaded application. The *manager/workers* in which the manager is responsible of input data and assigns work to others threads (workers). In a *pipeline* model, the task is divided into many sub-tasks. Each of these sub-tasks is concurrently handled by a different thread. Finally, the *peer* model is similar to the *manager/workers* one, except that the manager participates to the computation of the algorithms after creating a pool of the threads.

When IO process sends data to cluster process, IO and cluster process open communication objects on *WRONLY* and *RDONLY* mode respectively (see figure 7.4).

Inside clusters, the *pthread* library allows to create a multi-tasks program. In IO cluster, we consider that the multitasking application is executed on a single-core. In this case, parallel execution of tasks is virtual. Thus, it is therefore up to the operating system (RTEMS) on which the application runs to use its internal mechanisms to manage the execution of the various tasks of the application. However, in internal cluster we have a *true* parallelism; each core is dedicated to a single task.

In a cluster, a *main* thread always running on core 0, can create up to 15 others threads to run on other cores. The threads creation in one MPPA-256 cluster is described in Figure 7.4. These threads share the same memory. Classical synchronization like mutex, semaphore, synchronous barrier, conditional variable are used to synchronize concurrent threads.

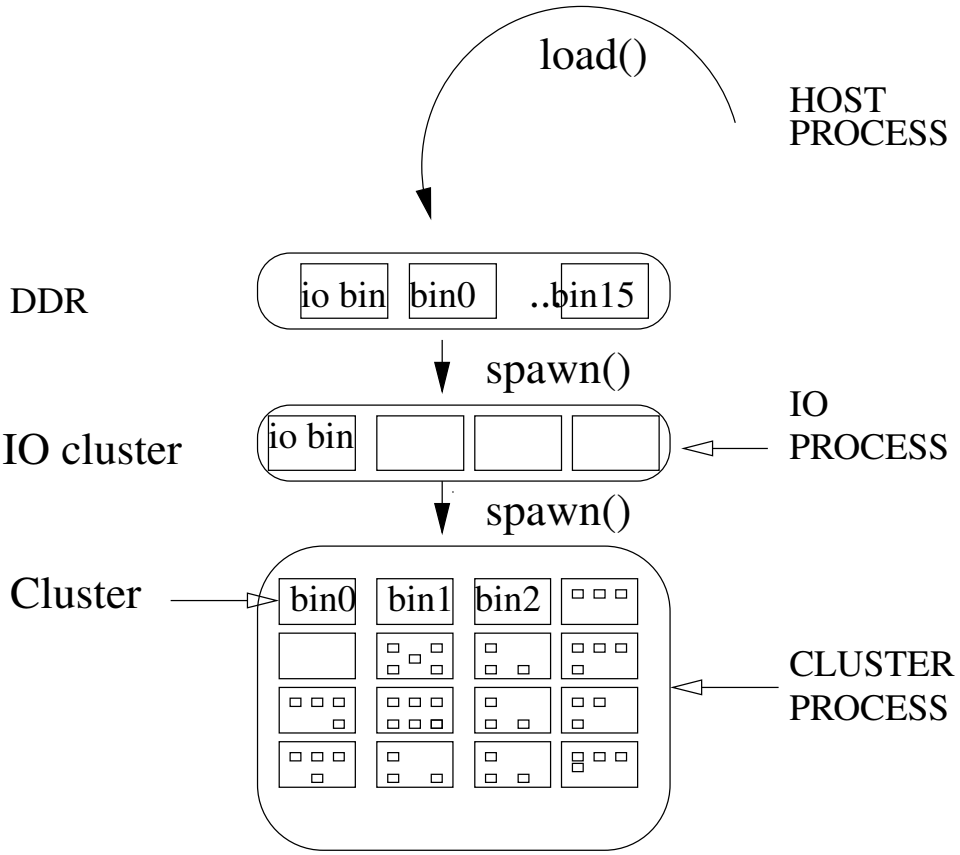


Figure 7.5: Binary Spawn on the MPPA-256

The MPPA-256 POSIX programming level operates as follows (refer to Figure 7.5): the HOST process loads a multi-binary executable on the MPPA-256 external DDR. Then the HOST process launches the IO executable and runs it on the IO Cluster by doing a `spawn()` operation. When executed in the IO Cluster, the `spawn()` function runs the executable code on the processing clusters. It is not possible to spawn executable code between processing clusters. Processes can send data using *argc, argv*.

7.4 Conclusion

The aircraft becomes a real information system. The avionic on-board integrates more and more functions. The first on-board computers improvement in term of size, weight and power is offered by the shift from federated architectures to IMA, thanks to the integration of multiple applications on the same computer.

Many-core processors reached the embedded systems market to address the high performance demanding functions and to improve the performance per watt thanks to the resources sharing between computing clusters and cores. They offer many opportunities to design modern avionic computers. The computation power offered by such processor allows to implement more resources consuming functions.

This gain of performance comes with several challenges due to many contention points in shared-resources. The main concern when designing an on-board function is not only to achieve the high performance demand but also to precisely determine the Worst Case Execution Time (WCET) for any task. According to [1], the processing timing analysis based-on measurement on the target has been demonstrated to be generally unsafe. The suitable practice is the static analysis based on modeling the temporal

behavior of the hardware. This estimation should consider all the sources of interferences in a many-core processor (cache, memory, bus, NoC, DDR). Then, the generic response time estimation of task τ_i running on a many-core processor is calculated as follows ([43]):

$$R_i = wcet_i + I^{PROC}(i, x, R_i) + I^{BUS}(i, x, R_i) + I^{DRAM}(i, x, R_i)$$

where $wcet_i$ is the worst case execution time in isolation time of task τ_i; x is the core where the task τ_i is running; $I^{PROC}(i, x, R_i)$ is the interference due to the preemption of a task which has a priority higher than τ_i; $I^{BUS}(i, x, R_i)$ is the interference on the bus depending on the mathematical model of the bus arbiter and finally $I^{DRAM}(i, x, R_i)$ is the interference due to DRAM refreshes.

Because of MPPA-256 many-core processor has predictability mechanisms since the design (memory banks, NoC reservation mechanisms). It offers spatial isolation by partitioning the shared-memory into 16 banks. Each bank has a separate bus arbiter. The MPPA-256 also guarantees a bounded traversal time on the NoC. Theses characteristics make the MPPA-256 a good candidate to estimate a tight WCET. $I^{PROC}(i, x, R_i) = 0$ for a static non-preemptive scheduler (only one thread per core).

Many avionic applications have real-time and safety requirements and must obey to strict certification processes. The certification approach requires to prove the correctness temporal behavior of the applications even if there are interferences. A study about the certification of the MPPA-256 has shown that its characteristics allow to reduce the number of interferences and thus the certification effort because each interference has to be identified and mitigated.

Chapter 8

Preliminary Experimental Results

The on-board Health Monitoring System we want to build in the scope of this thesis should be able to compute real-time indicators of the HMS using a many-core processor and to raise alerts to the operator crew.

The on-board architecture of the HMS is depicted in Figure 8.1. It is composed of a sensor board (sensors associated with an analog-digital converter) which is connected to a data acquisition unit. The acquisition unit provides a digital input flow to the MPPA-256 through a PCIe bus. Finally, the health indicators are sent to the display unit through a PCIe bus. Recall, that MPPA-256 has two PCIe interfaces (Section 7.3.2). One PCIe bus can be used both for the input and output flows. Otherwise, the first one can be dedicated to the input flow (raw vibration samples) and the second one to transmit the health indicators to the display unit.

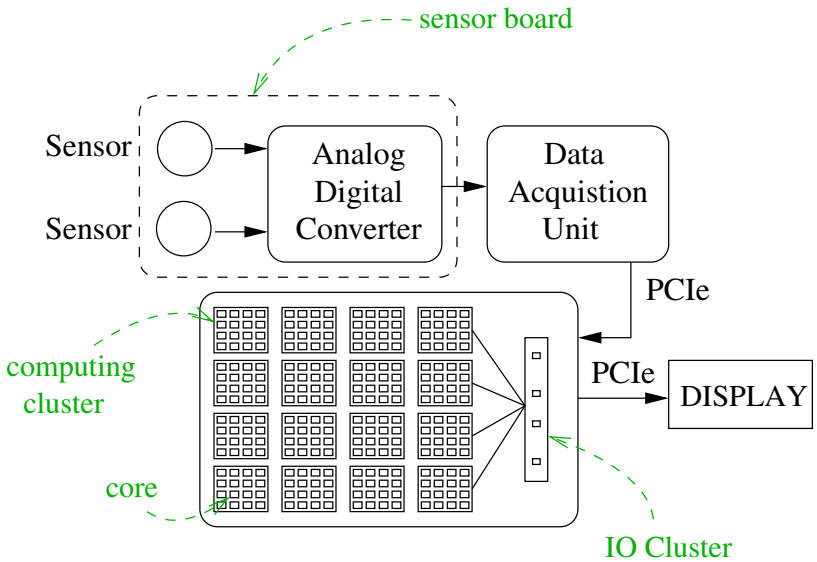


Figure 8.1: Mapping on the MPPA-256

However, our case study system in Figure 8.2 is not embedded in the helicopter yet. In our experiments, we replace the sensor board by the raw vibration data acquired during the flight. These files are stored in the Host memory. The display unit is also replaced by the *Host PC*. The Host PC and the MPPA-256 communicate through a PCIe bus.

This chapter covers preliminary experiments before deploying the HMS function on the MPPA-256.

These experiments are the following:

- *potential bottleneck on the IO cluster*: the IO cluster bottleneck is investigated by focusing first on the transmission of raw samples from the HOST to the IO Cluster (PCIe bandwidth) and then, from the IO Cluster to the Computing clusters by measuring the MPPA-256 latency. These two experiments aim at identifying a potential IO bottleneck when the HOST sends a packet of VIBRATION_CHUNK_SIZE samples (see Section 5.2, page 68) to the IO cluster. They do not involve any computation of indicators.
- *pipeline architecture for maximal throughput*: we build an example of a static mapping on the IO and computing clusters to maximize the throughput. This mapping separates the receiving and the sending functions in the HOST, IO cluster and internal cluster and allocates them to different threads.
- *workload estimation*: given a sampling frequency and a set of sensors, we evaluate the workload of one cluster of the MPPA-256 when computing *realistic* algorithms that are used to calculate indicators.

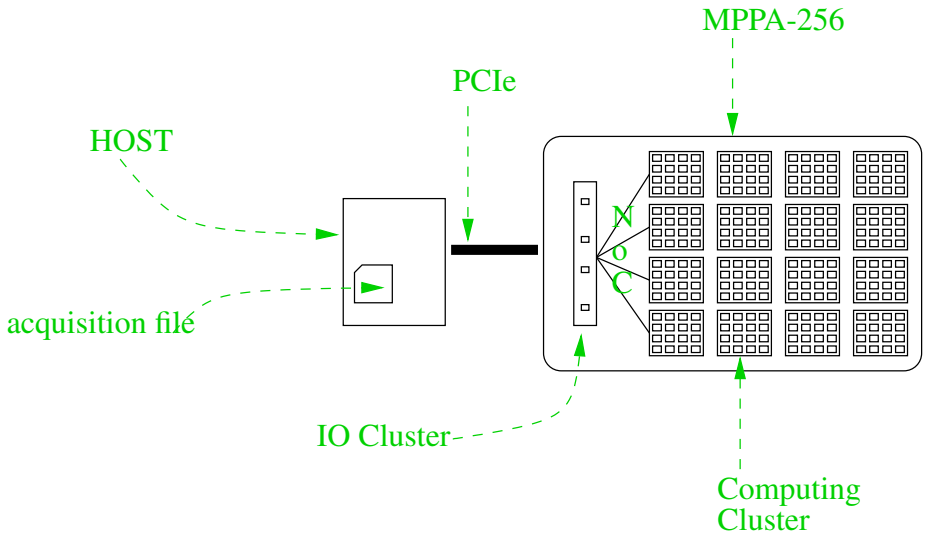


Figure 8.2: Overview of our case study system.

In addition to the preliminary experiments, we need a tracing mechanism in the HOST as well as in the MPPA-256 to measure the PCIe bandwidth, MPPA-256 latency and MPPA-256 workload. However, since the Host PC is not real-time, we perform our timing measurements on the MPPA-256 processor only.

8.1 Measuring Time on the MPPA-256

8.1.1 Requirements

For our experiments, we will need to perform timing measurement on the MPPA-256 target. We assume that our measuring sessions do not exceed 10 seconds. The number and the positions of timestamps depend on the measurements we desire to exploit. Some experiments may require only 2 timestamps while others much more. Finally, we need around 20 timestamps to characterize the full application as illustrated in Figure 8.13.

8.1.2 Existing timestamping tool on the MPPA-256

The MPPA-256 IO and computing clusters have both an internal clock running at 400 *MHz*. These clocks are mesosynchronous: they run at the same frequency but with different phase. The phase is around 100 cycles. This means that, when taking a timestamp T_0 in the IO cluster, and a timestamp T_1 in the computing cluster, the difference $T_1 - T_0$ cannot be more accurate than 100 cycles (250ns).

The Kalray software development kit (SDK) allows time measurements on a simulator of the K1 architecture (the cores of the MPPA-256). The default mode of the simulator offers maximum simulation speed but additional execution cycles due to processor pipeline stalls and cache memory accesses are not considered. The K1 simulator accuracy mode is not completely accurate because it is an abstraction model of the K1 processor. This model is a simplified version of the processor and offers a trade-off between simulation speed and accuracy. Consequently, we need to perform on-target measurements. Kalray also provides a tracing mechanism on the MPPA-256 processor.

This tracing mechanism is inspired from *LLTng* trace framework (see [65]) based on the tracepoint *instrumentation* of the linux kernel. The acquisition trace is installed as a linux kernel module which collects the timestamps. The existing tracing mechanism provides a large number of predefined tracepoints. When compiling an MPPA application, all of them are automatically available. However, they have to be activated by the user. The software developer adds specific macros in the source code to activate the tracepoints. The additional code size needed by a tracingpoint can go up to *320 bytes*.

We designed our lightweight tracing mechanism, by adding timestamps to the flows of data. The additional data should not change significantly the functional behavior of our application.

8.1.3 Lightweight timestamping tool principle

Each cluster has a Debug System Unit (DSU) offering a 64-bit counter for timestamping. For measuring up to 10s with the 2.5ns MPPA-256 clock period, we need a 32-bit counter. Since we consider of volume of data at least equal to 32768 bytes (refer to the latency experiment in 8.7), 20 additional timestamps correspond to an overhead of less than 0.3%.

In order to have a good timing precision and a less timing overhead, we use a system call based on *specific CPU register* to get the timestamp. This counter is set to 0 at board reset and incremented at each MPPA-256 clock tick. Its resolution is 1 cycle.

We design a timestamping tool that allows to transform a given dataflow to another one so that:

1. The number of tasks of the dataflow remains unchanged.
2. The dependencies between tasks remain unchanged.
3. *The only difference is the volume of data exchanged between tasks*.

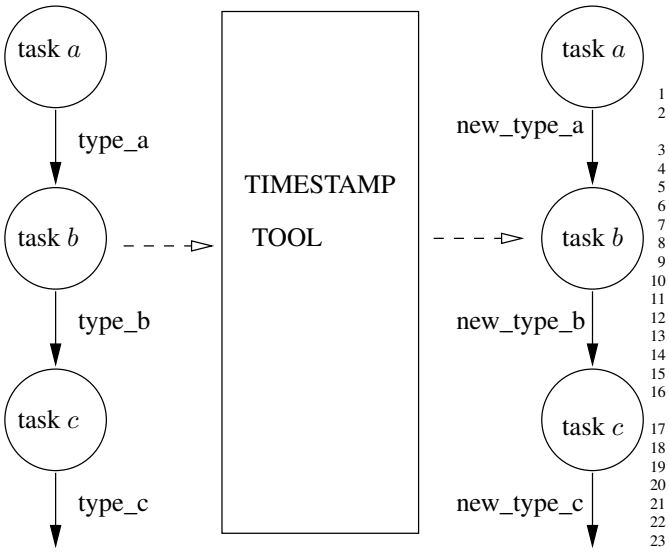
To illustrate the timestamps mechanism, we need to introduce the concept of *dataflow graph* D. Suppose we have an application that can be expressed as a collection of finite number of tasks $T(D)$. These tasks have dependencies and exchange data between them.

Definition: According to [66], a *dataflow graph* D is defined by

$$D = \{T(D), A(D)\}$$

Here, $A(D)$ contains dependencies of the form $a = (src(a), dst(a), type\_a)$, where $src(a), dst(a) \epsilon T(D)$ are the source, respectively the destination of task a, and $type\_a$ is the type of data transmitted from $src(a)$ to $dst(a)$. The dependencies between tasks imply a partial execution order (acyclic graph).

The execution of task a consists in reading the input data coming from $src(a)$, then computing the output data, which will correspond to the input of the destination $dst(a)$.



Listing (8.1) new data type after using the timestamps tool

```
1  typedef struct new_type_a = {
2  type_a output_task_a;   //
   output_task_a is an integer
3  int T0_a;
4  int T1_a;
5  }
6
7  typedef struct new_type_b = {
8  type_b output_task_b;  // output_task_b is a float
9  int T0_a;
10 int T1_a;
11 int T0_b;
12 int T1_b;
13 }
14
15 typedef struct new_type_c = {
16 type_c output_task_c;
   // output_task_c is an array
17 int T0_a;
18 int T1_a;
19 int T0_b;
20 int T1_b;
21 int T0_c;
22 int T1_c;
23 }
```

Figure 8.3: Transforming a Dataflow graph using the timestamping tool

To perform measurements on the MPPA-256, we will modify the type of data exchanged between the tasks. The principle of the timestamping tool is depicted in Figure 8.3. Task a reads from $src(a)$. In addition, task a computes algorithm using data coming from $src(a)$ and sets a set of timestamps during its processing. These timestamps are typically set at the beginning (after reading input data) and at the end of its processing (before sending output data). Then, the former $type\_a$ previously defined is transformed into new_type_a which is a data structure containing $type\_a$ and all timestamps previously set by task a.

In general, task i reads the data new_type_i which corresponds to the processing results of $src(i)$ with $type\_i$ and the set of timestamps gathered by the task i .

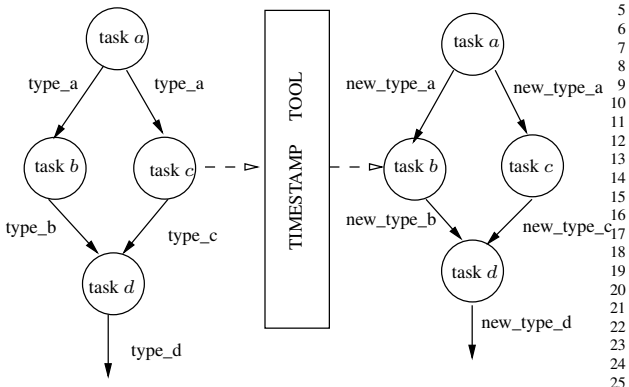
8.1.4 Timestamping example (case 1)

Consider the example depicted in Figure 8.3, where $T(D) = 3$. The dataflow graph is composed of 3 tasks a, b and c. In addition, we suppose that tasks a, b and c produce respectively an output of type int, $float$ and $array$. Then $type\_a = int$, $type\_b = float$ and $type\_c = array$. Finally each task sets respectively 2 timestamps $(T0\_a, T1\_a)$, $(T0\_b, T1\_b)$ and $(T0\_c, T1\_c)$.

Listing 8.1 corresponds to the declaration in C language of the new types $(new\_type\_a, new\_type\_b, new\_type\_c)$.

8.1.5 Timestamping example (case 2)

We consider a *dispatch-gather* dataflow depicted in Figure 8.4, the task a produces a data of type $type\_a$ which will be used by 2 other tasks (tasks b and c). Then the tasks b and c use the data of type $type\_a$, run in parallel and respectively produce a data of type $type\_b$ and $type\_c$. Finally, the task d waits for the completion of tasks b and c and produces a data of type $type\_d$.



Listing (8.2) new data type after using the timestamps tool

```
1  typedef struct new_type_a = {
2  type_a output_task_a;
3  int T0_a;
4  int T1_a;
5  }
6
7  typedef struct new_type_b = {
8  type_b output_task_b;
9  int T0_a;
10 int T1_a;
11 int T0_b;
12 int T1_b;
13 }
14
15 typedef struct new_type_c = {
16 type_c output_task_c;
17 int T0_a;
18 int T1_a;
19 int T0_c;
20 int T1_c;
21 }
22
23 typedef struct new_type_d = {
24 type_d output_task_d;
25 int T0_a;
26 int T1_a;
27 int T0_b;
28 int T1_b;
29 int T0_c;
30 int T1_c;
31 int T0_d;
32 int T1_d;
33 }
```

Figure 8.4: Transforming a Dispatch-Gather Dataflow graph using the timestamping tool

The main difference compared to the previous example is that: during the dispatching step, each of the two tasks running in parallel has a copy of the timestamps set by task a. Finally, the task d gathers the timestamps set by the previous tasks.

8.2 Experiment focusing on data arrival

8.2.1 PCIe bandwidth

PCIe

The *PCIe* technology is a serialized point-to-point interconnect protocol developed by Intel in 2004 ([67]). It may have one or more data transmission *lanes*. Each lane has two pairs of wires, one for sending and another one for receiving data. According to [68], each lane is also an independent connection between the PCIe controller and the expansion card. The bandwidth scales *linearly* with the number of lanes. For instance, when considering the same PCIe generation, a PCIe with *sixteen* lanes connections has *twice* the bandwidth of a PCIe with *eight* lanes.

Gigatransfers per second (GT/s) refers to the number of operations transferring data that occur at each second for a given channel. The *Gigabits* transfer rate is calculated by multiplying the Gigatransfers rate by the channel width.

According to [69], the maximum transfer rate of generation 1 (Gen 1) PCI Express is *2.5 GT/s*; Generation 2 (Gen 2) PCI Express is *5.0 GT/s* and Generation 3 (Gen 3) PCI Express systems is *8.0 GT/s*. These rates indicate the *Gigatransfers* per second (GT/s) per lane in a single direction.

In fact, PCIe loses an amount of its maximum theoretical bandwidth due to physical overhead associated with electronic transmissions. The PCIe (Gen 1) and (Gen 2) use the same encoding protocol: 8B/10B which means that each transmission of 8 bits costs 10 bits. They lose 20% of their bandwidth

on overhead. Nevertheless, the PCIe (Gen 3) has a more sophisticated and efficient encoding scheme: 128B/130B which has only an overhead of 1.54% instead of 20%. The effective bandwidth for various PCIe generations and link widths is calculated as follows:

$$\text{GEN1: Effective Bandwidth}(GB/s) = \frac{2.5 \times 2(\text{two directions}) \times \text{Lanes width}}{10\text{bits/byte}}$$

$$\text{GEN2: Effective Bandwidth}(GB/s) = \frac{5 \times 2(\text{two directions}) \times \text{Lanes width}}{10\text{bits/byte}}$$

$$\text{GEN3: Effective Bandwidth}(GB/s) = \frac{8 \times 2(\text{two directions}) \times \text{Lanes width}}{130/128}$$

Table 8.1 shows the effective theoretical bandwidth in *full-duplex* of different PCIe generations with various link widths.

	Lane width			
	x1	**x2**	**x4**	**x8**
PCIe GEN 1 (GB/s)	0.5	1.0	2.0	4.0
PCIe GEN 2 (GB/s)	1.0	2.0	4.0	8.0
PCIe GEN 3 (GB/s)	2.0	**3.9**	7.9	15.8

Table 8.1: Effective PCIe bandwidth in full-duplex with various lane widths.

PCIe bandwidth experiment

The purpose of this experiment is to measure the PCIe bandwidth. In the scope of thesis, the *third* PCIe generation (GEN 3) is used as the interface between the HOST and the IO cluster. The number of lanes is equal to 2. Therefore, the expected maximum bandwidth in full duplex is 3.9 GB/s (i.e. *1.95 GB/s* in one direction) and is illustrated in Table 8.1.

Our experiment consists in one IO thread (*IOsenderToHOST*) and 1 HOST thread (*HOSTreceiver*). The computing clusters are not considered in this experiment.

The thread *IOsenderToHOST* sends data of different size to the thread *HOSTreceiver*. For each packet of data sent by the thread *IOsenderToHOST*, we measure the time that the thread *IOsenderToHOST* takes to send that packet including two timestamps (T0,T1). T0 and T1 are respectively set before and after sending data to the thread *HOSTreceiver*. This test is performed 100 times in order to observe the jitter.

Knowing the volume of data and the duration, we plot the PCIe bandwidth in one direction in Figure 8.5. The maximum bandwidth in one direction is *1.88 GB/s*. This value is consistent with the theoretical bandwidth of a PCIe GEN3 x2 in one direction equal to 1.95 GB/s. In addition, the jitter observed in Figure 8.5 is due to the HOST which is not real-time.

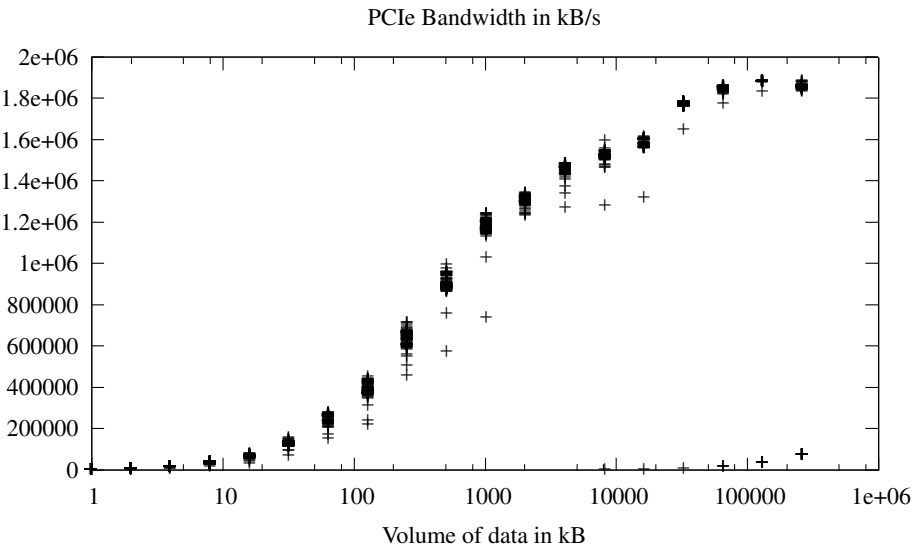


Figure 8.5: GEN3 PCIe (with 2 lanes) bandwidth experiment with data of different size sent by the IO thread. The experiment is repeated 100 times for each packet of data sent by the IO thread to the HOST thread.

Recall, the HMS function has a few dozen of sensors and the maximum sampling frequency (fs) is $31kHz$. Then, the required bandwidth is less than 5 MB/s. The bandwidth capability offered by the PCIe GEN3 is sufficient to transfer the input flow. However, the 5 MB/s will not be transferred once to the MPPA-256 due to the limited amount of memory inside each computing cluster (2MB). The size of the packet of samples VIBRATION_CHUNK_SIZE transmitted from the acquisition card to the MPPA-256 is determined by the latency experiment detailed below.

8.2.2 Dimensioning Experiment: MPPA-256 Latency

We focused on inputs, since the current MPPA-256 implementation is limited to using only one IO core of the IO Cluster, which can cause a latency issue. We use only one processing cluster among the 16 physically available. Then, there is no computation in the processing cluster.

In order to observe the end-to-end latency, we made an experiment where data are just received by the MPPA-256, and sent back to the host.

We implemented a *synchronous* dataflow architecture: it consists in 3 threads running respectively on the HOST PC, an IO Cluster and an Internal Cluster as described in figure 8.6.

The HOST thread sends samples to the IO cluster thread which in turn forwards them to the internal thread of the cluster. There is no algorithm implemented in the internal cluster. Then the internal cluster thread sends back data to the IO cluster thread, which finally transfers it to the HOST thread. The host thread must receive data before it sends another sample. We measure the MPPA-256 latency: the time it takes for one bit to make a complete round trip through the IO and the internal clusters.

The experiment is performed for increasing sizes of data-chunks, and 100 times for each size. Figure 8.7 shows the MPPA-256 latency in μs for various data sizes in bytes. Figure 8.7 shows that the MPPA-256 latency remains less than 200μs for sizes comprised between 4 and about 8192 bytes. Beyond 32768 bytes the MPPA-256 latency increases linearly. *Then, in this chapter, we choose a volume of data 32768 bytes which corresponds to a latency of $784\mu s$.*

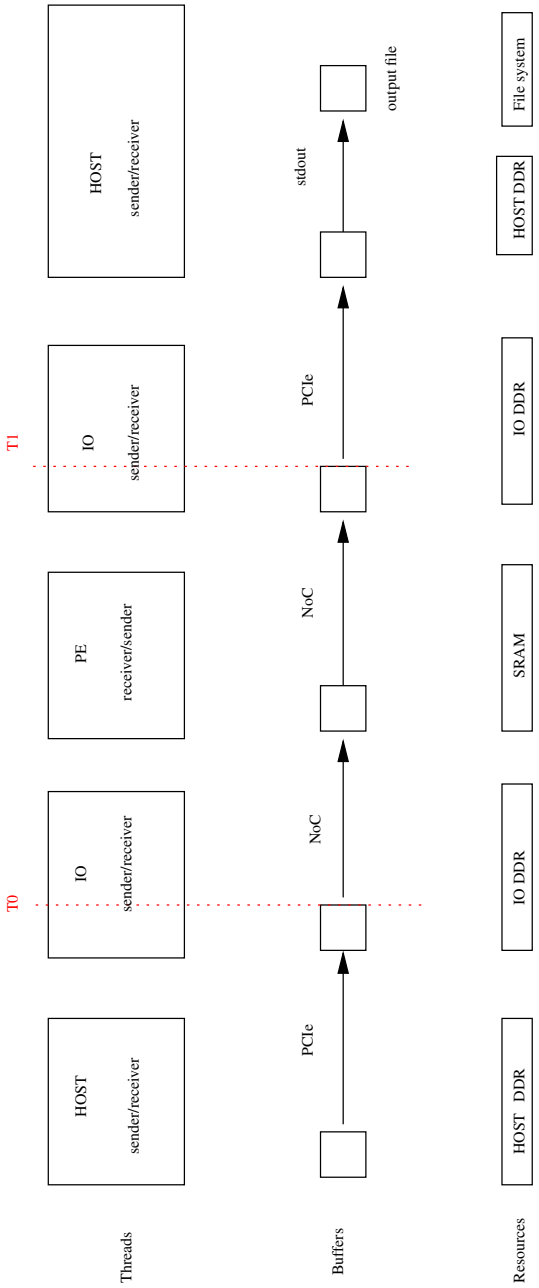


Figure 8.6: Synchronous Dataflow Architecture. TO and T1 are both set by the IO thread: T0 after receiving data coming from the HOST and T1 before sending data to the HOST.

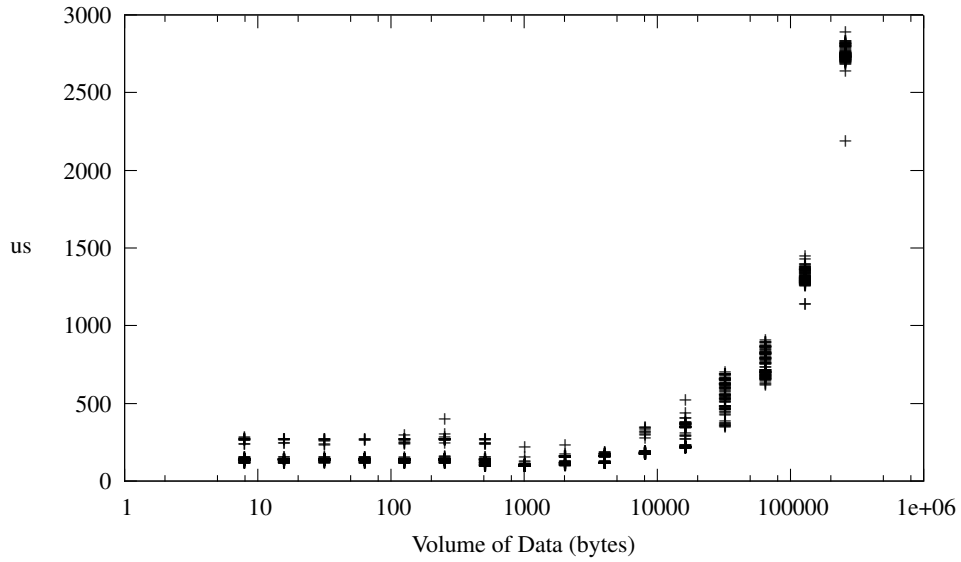


Figure 8.7: MPPA Latency. The x-axis is linear and indicates the volume of data and y-axis (logarithmic scale) is the MPPA-256 latency in us. For a given packet of samples, the latency is measured 100 times.

This experiment allows to choose the minimum size of packets to send to the MPPA-256: VIBRATION_CHUNK_SIZE = 8192 samples (each sample is a float: 32 bits). On the other hand, the maximum size should be determined by taking into account real applications in which there is a significant processing duration and the MPPA-256 memory constraint.

Assume that for a data size d, the processing time is defined by a function $f(d)$ and the corresponding latency $l(d)$. d should be defined such that $f(d) \gg l(d)$. It means, the HMS function should spend more time processing the indicators than waiting for the arrival of new data.

8.3 Pipeline architecture for maximal throughput on MPPA-256

We focus on inputs here, and the constraint of computing health-indicators sufficiently fast with respect to the volume of data determined by the input frequency and the number of sensors.

Recall, the MPPA-256 is made of a first stage containing the 4 input/output (IO) cores, and a set of 16 clusters of 16 cores each (called *processing elements, or PEs*). Assume that the HMS function has s sensors. Since sensors are functionally independent of each other, we can decide that each of the 4 cores of the IO Cluster manages $s/4$ sensors. Figure 8.8 shows the distribution of sensors across IO cores in this case.

However, the current MPPA-256 implementation is limited to only one IO core being used by the SMP scheduler of the RTEMS operating system. We could parallelize the work logically with threads on this unique active core, but this would not improve timing. Figure 8.9 shows the limitation with the current MPPA-256 implementation.

In this static assignment, our code would decide before execution which cores manage which tasks, which refers to a *Bounded Multi-Processing (BMP)* as illustrated in [70]. One objective of the mapping will be to minimize end-to-end application execution time, i.e., the time it takes for one sample packet to travel from source to destination. In our case, this duration shall be lower than the period of sampling in order to compute the sensors samples in real-time. The logical structure of the work to be done is shown in Figure 8.10: there are three steps, that have to be performed in sequence because each part depends

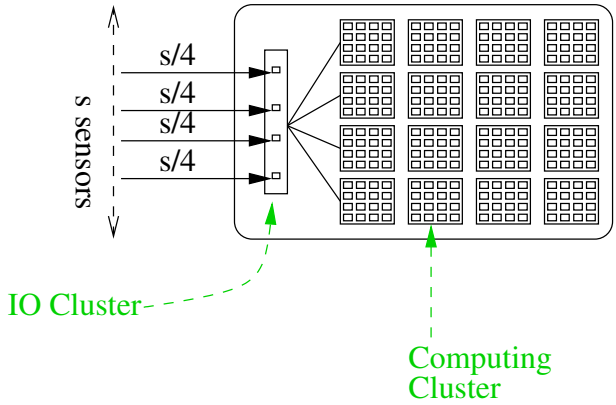


Figure 8.8: Distributing Sensor Samples on the cores of the IO Cluster

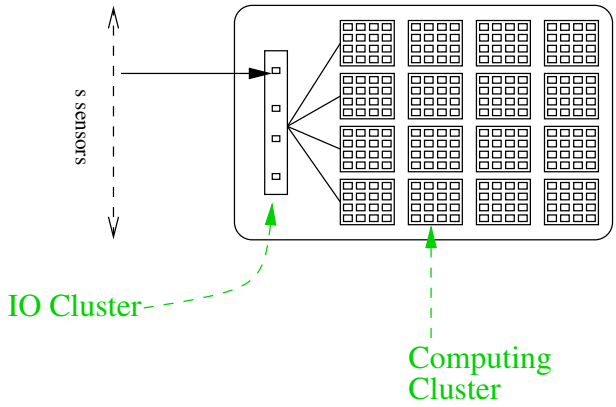


Figure 8.9: Using only one core of the IO cluster for the Sensor Samples

on data output from the previous part (*dispatching, processing, gathering*). Dispatching is required to transform an input sample vector (which contains one sample of all sensors), into a set of vectors for each sensor to be processed independently. Then processing can be applied on each sensor in parallel. Finally, all processing outputs are gathered to be displayed and stored on a device. We evaluated two choices:

- Allocating dispatching and gathering tasks to the IO cluster, and processing to the PEs of one computing cluster.
- Allocating dispatching, processing and gathering to the computing cluster. The IO cluster serves only for routing packets from/to the HOST processor.

8.3.1 *Dispatching* and *Gathering* in the IO cluster

Figure 8.11 illustrates this choice. In this configuration, the IO cluster manages both *Dispatching* and *Gathering*, implemented as two tasks running on the same core. *Processing* the data for each sensor is allocated to one of the PEs of a computing cluster.

The MPPA-256 architecture is such that the IO cluster accesses only DDRAM, and the computing clusters access only internal shared SRAM. The SRAM has a lower latency than the DDR and is not shared with other clusters and IO devices. In each cluster, there is a single DMA controller bound to

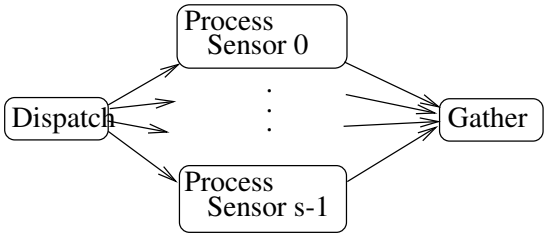


Figure 8.10: Functional structure

the sending thread. Obviously, the software overhead to call the send or receive packet services and use of the DMA resource, leads to a better performance when sending all data in a single packet, than in multiple smaller packets. Some of our experiments confirmed that sending one sample packet by sensor is less efficient than sending a single packet that contains all sensors samples. Each IO core is associated with one DMA controller. It is useless to send packet sensor data one by one because only one DMA is available. Thus, it is more efficient in terms of cycle duration using one POSIX system-call to send all sensors gathered in one stream rather than sending them one by one.

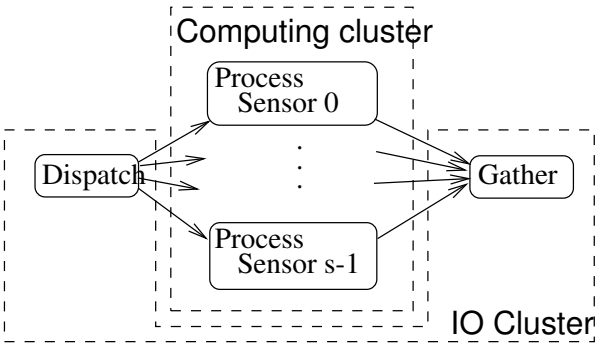


Figure 8.11: Dispatching and Gathering mapped on IO Cluster

8.3.2 *Dispatching* and *Gathering* in the computing cluster

Figure 8.12 illustrates this choice. Each function (Dispatching, Gathering and Processing sensor x) runs on one PE of the computing cluster.

We intend to ensure *load balancing* through static assignment i.e., to keep all cores busy as much as possible and avoid overloading of any single resource (NoC route, DMA, core). We need the next data to be processed to be available at the moment when the PEs work on the current one. Assume we have s sensors, a algorithms to compute (the details of "processing" sensor x) and p processing elements. We have two choices to distribute calculations over processors. We describe each choice in details below.

Parallelizing the algorithms

It means dedicating one core among the p processing elements to each particular function among the f functions to compute data coming from all sensors. For instance, in equation 8.1, $\text{FFT}, \text{FFT}^{-1}, Remove$ and RMS are examples of functions used to compute indicators. The functions involved in the HMS function to compute indicators exchange data: outputs produced by one of them are often re-used by another. For instance, the RMSR indicator calculated as follows:

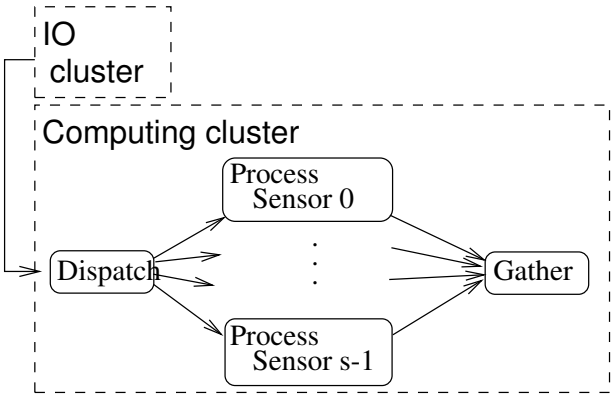


Figure 8.12: Dispatching and Gathering mapped on the computing cluster

$$\overbrace{\text{RMS}(\overbrace{\text{FFT}^{-1}\,(\mathit{Remove}(\underbrace{\text{FFT}(\mathit{AverageSignal})}_{1},V))}^{3})}^{4} \underbrace{\qquad\qquad}_{2} \tag{8.1}$$

$AverageSignal$ denotes the average of all monitored shaft revolutions and V is the set of harmonics to remove from the spectrum.

It requires 4 functions: Fast Fourier Transform, Remove harmonics, Reverse Fast Fourier Transform and RMS. If functions are allocated to distinct PEs, this involves a synchronization overhead, which depends on the number f of functions; the communication overhead will also be significant.

Parallelizing the sensor data

It means allocating one core among the p processing elements to one sensor among the s sensors, and to compute all a algorithms for the same sensor on that core. Sensors are functionally independent, thus threads can run without needing any data exchange. The *load balancing* seems to be perfect, since each core computes several algorithms sequentially on one sensor data. The cores do not need to communicate. However, the cores must receive data coming from the IO cluster and transmit their results to the IO cluster. If a thread, besides its work to process algorithms, is in charge of reading or writing data to the cluster IO, this creates a *poor load balancing*, because the latter thread is always busy while others are waiting.

To avoid this problem, we use two more cores. The first one is dedicated to the reading of the data coming from the IO cluster (Dispatching) and the preparation of workers inputs. The other is in charge of transmitting the result of the workers to the IO cluster (Gathering) (see Figure 8.12). In one computing cluster, the maximum number of cores usable to compute algorithms is therefore 14, among the 16 cores that are physically available. We will call these cores *workers* in the sequel.

8.4 Experimental settings

8.4.1 Dataflow with timestamps

Figure 8.13 describes the reference architecture of our experiments. This software architecture is composed of several threads that can be grouped into 3 families.

- The first family is the threads running on the HOST: HOSTSENDER and HOSTRECEIVER. The HOSTSENDER thread is used to mimic the acquisition unit. It provides a chunk of vibration samples to the MPPA-256. In addition, HOSTRECEIVER thread is used to receive the health indicators computed by the internal clusters.
- The second thread family is the IO threads: IORECEIVERFROMHOST and IOSENDERTOHOST. They act as a link between the HOST and the computing clusters. In other words, these 2 threads allow to forward the vibration data and tops coming from the HOST to the computing clusters and also transmit the indicators to the HOST.
- Finally, the third thread family refers to the computing threads. They are called *workers*. This name refers to the threads that are responsible for computing the indicators. The number of workers depends on the experiment. There can be one for a sequential implementation or many to explore parallelism.

Each of these threads can be decomposed into sub-functions.

- IORECEIVERFROMHOST thread in figure 8.13 is composed of 2 main functions: **mppa_read** and **mppa_aio_write**. The prototype of the function **mppa_read** is as follows:

```
1 Int mppa_aio_read(mppa_aiocb_t * aiocb);
2
3 /**It returns:
4  0 on success,
5 -1 on error.
6 */
```

mppa_read is a *synchronous* function (blocking function) of the *MPPA IPC API* which allows to read the data exchanged between HOST and IO. This function can be used both by the IO when it reads data from the host or vice versa. It reads **size** bytes from the PCIe file descriptor **fd** and stores them in a buffer which address starts at **base**. When the mppa_read function is called by the thread HOSTreceiver, the read data are stored in the *HOST DDR*. On the other hand, data are stored in the DDR of the IO cluster when it is called by the IO thread IORECEIVERFROMHOST. To ensure that all data are read, a comparison between the size of read data and the size of expected data is often performed after calling **mppa_read** function. This comparison is usually set by using the *assert* function from C standard library.

mppa_aio_write is an *asynchronous* function of the *MPPA IPC API* which allows to exchange data between clusters. This function can be used both by the IO cluster to send data to the computing cluster or vice versa. Its prototype is the following:

```
1 Int mppa_aio_write (mppa_aiocb_t * aiocb);
2
3 /**
4 * Aiocb Pointer to a mppa_aiocb_t object.
5 * Return 0 on success, -EAGAIN if not enough resources, else -1.
6 * In case of error, the errno variable is set.
7 * /
```

mppa_aiocb_t is an object that allows to initialize the route of transmitted data on the NoC. This object also permits to specify the buffer to send as well as its size in bytes. It returns:

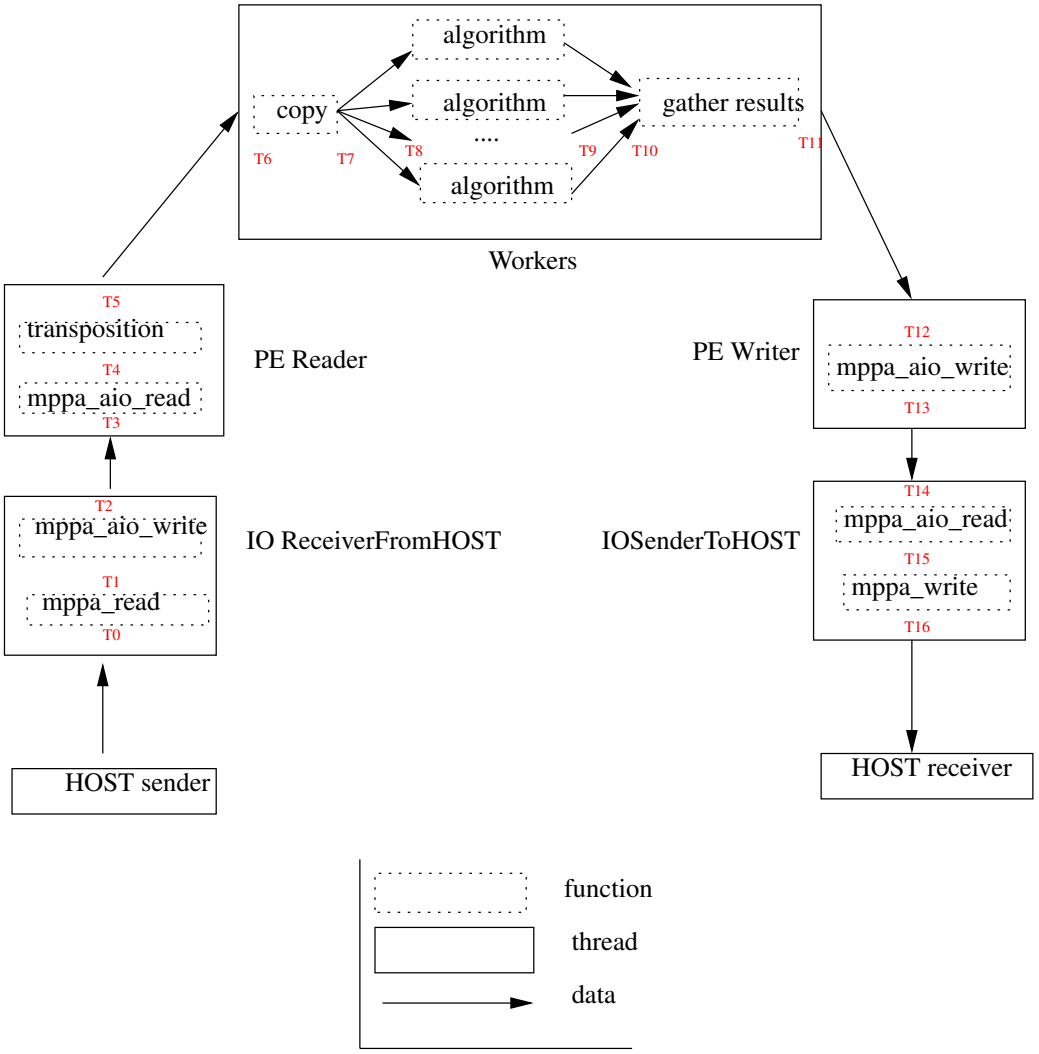


Figure 8.13: Dataflow diagram with timestamps

0 on success,

-EAGAIN if not enough resources,

else **-1**.

An error occurs when DMA threads are not available. In this case, *errno* is set to *EAGAIN*.

- PE READER thread in figure 8.13 is composed of 2 main functions: **mppa_aio_read** and **transposition**.

  1. **mppa_aio_read** is the symmetric function of mppa_aio_write. It allows an asynchronous reading of the data exchanged between clusters. This function can be used both by the IO to read data coming from the computing cluster or vice versa. Its prototype is the following:

```
1 Int mppa_aio_write (mppa_aiocb_t * aiocb);
2
3 /**
```

```
4 * Aiocb Pointer to a mppa_aiocb_t object.
5 * Return 0 on success, -EAGAIN if not enough resources, else -1.
6 * In case of error, the errno variable is set.
7 * /
```

Like the mppa_aio_write function, mppa_aiocb_t is an object that allows to initialize the route that our data will take on the NoC to go from the transmitter to the receiver. This object allows to specify the buffer to send as well as the size of buffer to read. When the PE READER calls this function, the data sent by the IO is now available in the internal computing cluster. The data read by the *PE Reader* thread are stored in the shared memory of the MPPA-256(SRAM).

2. **Transposition**:

   The data received by the READER PE thread comes from the acquisition unit and go through the IO cluster. The data keep their structure in cluster internal memory as they were in the acquisition unit. They move from the acquisition unit to the IO DDR to end up in the shared memory of the internal cluster. Recall, for each sampling period T_1 (we suppose that all accelerometers have the same sampling frequency fs), the acquisition unit builds a sequence of tuples:

   $1 * T_1 \Rightarrow (v_1^1, v_2^1, ..., v_N^1, TOP_1)$
   $2 \times T1 \Rightarrow (v_1^2, v_2^2, ..., v_N^2, TOP_2)$
   $3 \times T1 \Rightarrow (v_1^3, v_2^3, ..., v_N^3, TOP_3)$
   .....
   $N \times T1 \Rightarrow (v_1^N, v_2^N, ..., v_N^N, TOP_N)$

   where T_1 ,v_i^j and TOP_j represent respectively the sampling period, the value of the vibration of accelerometer i and the top at $j \times T_1$

   These samples are located at contiguous memory addresses in the acquisition unit as well as in the SRAM memory. More precisely, vibration data and tops are stored in the following order in the memory: $(v_1^1, v_2^1, ..., v_N^1, TOP_1)$ $(v_1^2, v_2^2, ..., v_N^2, TOP_2)$ $(v_1^3, v_2^3, ..., v_N^3, TOP_3)$ ... $(v_1^N, v_2^N, ..., v_N^N, TOP_N)$

- WORKERS Workers in figure 8.13 are composed of 3 main functions: *copy, algorithm* and *gather results*. These 3 functions are both data producer and consumer.

  1. **Copy** performs a copy of data produced by PE READER. This function releases the PE Reader thread after performing the copy. Thus, the PE READER thread can read new data coming from the IO Cluster in parallel of the processing of the previous data.
  2. The **algorithm** function is the processing part. It takes as input the data produced by the **copy** function and gives its results to the *gather results* function. Each worker reads its data in the shared memory and writes the result of its computation into the shared memory.
  3. The **gather results** function is used to group the results of all workers into one data structure. It prepares the data that will be sent to the IO cluster.

- *PE Writer* is the interface between the internal and IO clusters. It sends the results of the computation from internal cluster memory to IO DDR through the NoC.

- IOSENDERTOHOST is composed of 2 main functions: **mppa_aio_read** and **mppa_write**. These 2 functions read respectively data coming from the internal cluster and forward them to the HOST.

Data read from PE READER thread are stored in the IO DDR memory and transfer to the HOST DDR.

- HOSTRECEIVER is the last stage in the indicator computation chain. It receives the results of the indicators of the HMS. In addition to these indicators, it gathers a set of timestamps set by all threads during the computation chain.

The timestamping tool described in Paragraph 8.1.3, page 109, is used to set timestamps in the computation chain. At each thread, the timestamps are positioned. The number of timestamps positioned per thread may be variable. The main goal is to position timestamps according to the measurement we need to exploit. For example, in figure 8.13, T1-T0 measures the duration of the primitive mppa_read; T3-T2 is the NoC traversal duration (this measurement has an imprecision of 100 cycles) and T9-T8 measures is the processing time of the health indicators.

8.4.2 Synchronization mechanism

Once the data arrive into the internal cluster memory (SRAM), classical synchronization mechanism like barrier, conditional variable, mutex, semaphore are used.

PE reader, *PE Writer* and *Workers* threads exchange data through the 2MB shared-memory. Workers are both consumers and producers of data because they consume data produced by the *PE reader* and then produce indicator results. We must implement mechanisms to ensure the coherency of exchanged data between *PE reader* and *Workers* and between *Workers* and *PE Writer*. We use C11 atomic built-ins that bypass the cache to implement a *producer/consumer* synchronization concept (refer to Figure 8.14).

8.5 Experiment focusing on MPPA-256 workload estimation

The following experiments represent a logical sequence of the previous . The previous experiment allows us to find the minimum size of packets to be sent to the MPPA-256. However, this is not a realistic scenario and does not resemble to the real implementation of our application because no computation is made in the internal clusters.

In the following experiments, the workers perform a realistic computation of some signal processing algorithm, which approximates the kind of algorithms computed in the real application.

The aim of this series of experiments is to find for a given sampling frequency and number of sensors, a computational load that the MPPA-256 can process without buffering input vibration data.

We will verify whether the behaviors of the PE reader, the PE writer and the 8 workers are well pipelined. Then we will study the ratio Data Transfert Duration/Processing Duration.

8.5.1 General Settings

The HOST process loads a multi-binary executable on the MPPA external DDR. Then the HOST process launches the IO executable and runs it on the IO Cluster by doing a `spawn()` operation. When executed in the IO Cluster, the `spawn()` function runs the executable code on the processing clusters. It is not possible to spawn executable code between processing clusters.

The samples received by the HOST are not directly recorded in a file for post-processing, because the latency of the file-sytem would impact the throughput. Samples are written on the standard output *stdout*. When running the code we redirect the standard output so that another HOST process reads the standard output buffer with a *pipe* and writes data on a non-volatile mass memory.

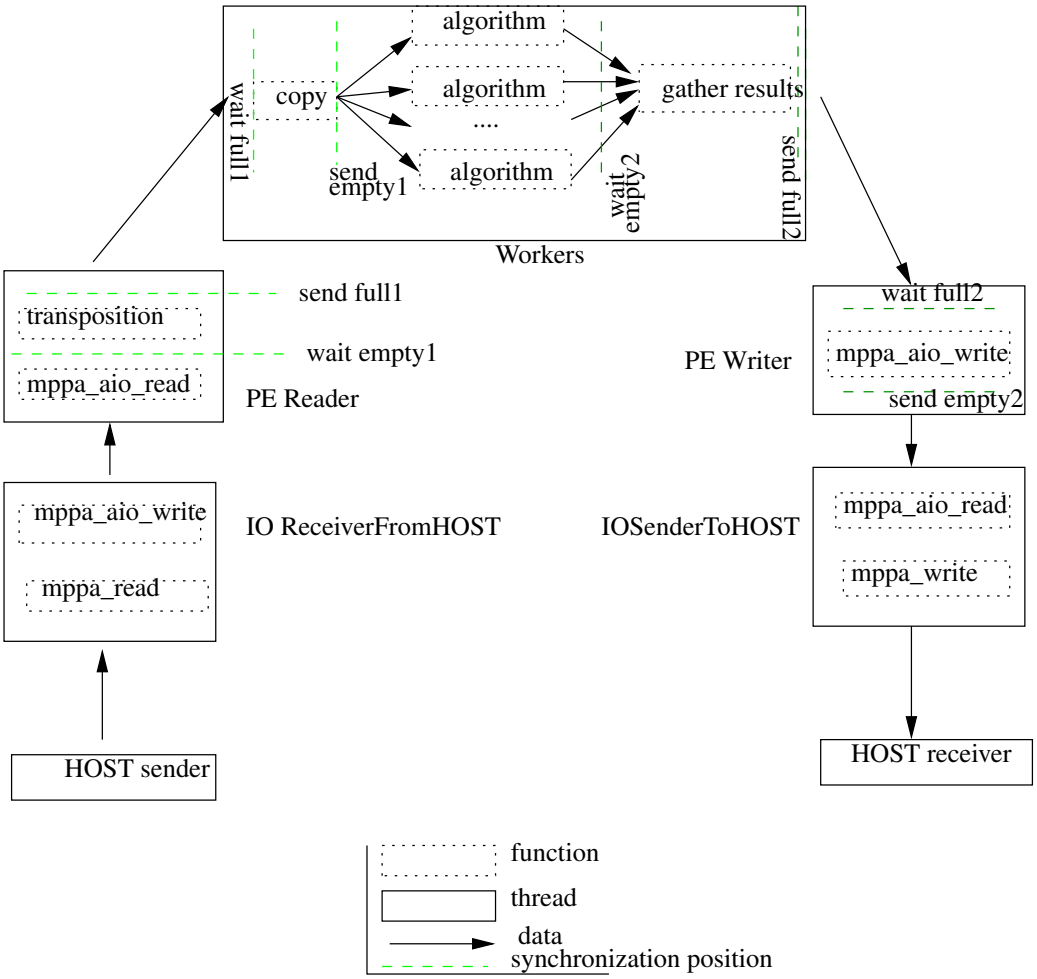


Figure 8.14: Dataflow diagram with synchronization positions

Data Architecture

In order to avoid the problems encountered with the abovementioned *synchronous dataflow* architecture, we separate *sending* and *receiving* (in the Host, IO and computing clusters) allocating them to different threads. The PE reader thread (See Figure 8.15) reads data of type *vector [N][S]* (recall the HMS function has S sensors and each sensor delivers N samples during each acquisition session) coming from the IO Writer; it produces `sensorsBuffer[S][N]` after the matrix samples *transposition*. All worker PEs access the `sensorsBuffer` data structure, using different indices. For instance, we pre-assign `sensorsBuffer[0]` to PE worker0, `sensorsBuffer[1]` to PE worker1, `sensorsBuffer[2]` to PE worker2 and so on. A worker processes an FFT and computes its Module using data from a single sensor. It writes its results in a shared buffer *Module* (see Figure 8.17).

Is it necessary to perform a transposition step?

The **transposition** function improves the performance of the core by decreasing the number of cache miss. Indeed, since the frequency gap between cores and memories is increasing, different optimization techniques are used to improve performance of the cores, in particular the temporal locality of the data. The *splitting* and *reordering* [71] techniques are used in *mono-processor* architecture to decrease cache miss. These techniques aim at improving cache performance by ordering related data on the same cache line. *Splitting* consists in grouping the data that are frequently used by the core. These data are partitioned into *hot* and *cold* group depending on access frequency. The *reordering* groups the data according to a certain logic. Data are reorganized such that related data are near each others in the cache. In our case study, the transposition function is used as a reordering technique to classify the data according to the type of accelerometer.

In *multi-core* environment, conventional *splitting* and *reordering* techniques are insufficient to reduce cache misses because processor performance may be degraded for other reasons. After the transposition operation, the vibration samples are arranged as follows in shared memory: the samples of the accelerometer 1, followed by the samples of the accelerometer 2, etc.

In fact, each core accesses *independent* data. For example, the core 1 will access the data of accelerometer 1 `sensorsBuffer[0]`. It is the same for the core 2 which will read the data of the accelerometer 2 `sensorsBuffer[0]`. However, the data transfers between the shared memory and each PE are performed per *cache line*. Which means that when the PE 1 tries to access the first vibration sample of accelerometer 1 (`sensorsBuffer[0][1]`). Then, core 1 will have in its private cache those of the other accelerometers too. Thus, each modification of the data by a given PE will lead to an invalidation of the cache of the other PEs, even though their values at others PEs are correct: *false sharing*.

Since the MPPA-256 requires to manage cache coherency manually, we ensure by alignment directives that the various samples of each accelerometer will not share cache lines, in order to avoid the need for this manual cache management. This operation is performed by using *attribute(aligned(0x20))*; 0x20 (32 bytes) represents the data cache line size.

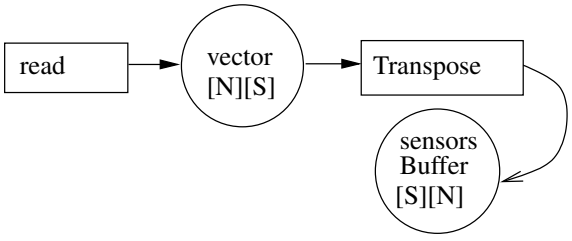


Figure 8.15: PE Reader Thread

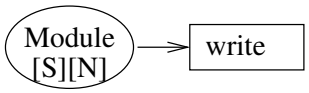


Figure 8.16: PE Writer Thread

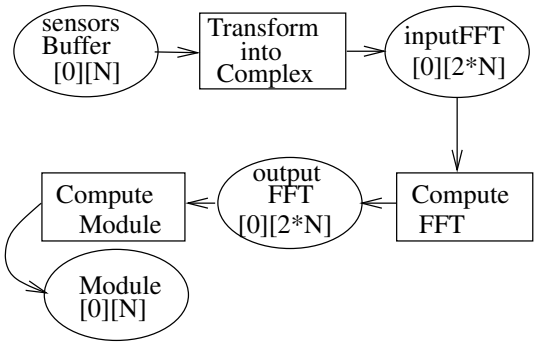


Figure 8.17: Worker0 Dataflow

`sensorsBuffer` is an array containing samples of all sensors. `sensorsBuffer` is treated as a one-dimensional array; it is arranged according to the *row order* by the C compiler. In others words, all data of sensor0 (`sensorsBuffer[0]`) come first, then all data of sensor1 and so on. Worker0 accesses `sensorsBuffer[0]` and these data should not be altered because `sensorsBuffer[0]`, `sensorsBuffer[1]`, etc., are independent.

However the workers may share cache lines. This is called the *false cache sharing* phenomenom. Since the MPPA-256 requires to manage cache coherency manually, we ensure by alignment directives that the various `sensorsBuffer[x]` will not share cache lines, in order to avoid the need for this manual cache management. This operation is performed by using `attribute(aligned(0x20));` `0x20` (32 bytes) represents the data cache line size. Consider the Figure 8.17: all workers access the data using different indices in `sensorsBuffer, inputFFT, outputFFT` and `Module`. Like `sensorsBuffer` all these data should be cache-line aligned.

The PE writer, the PE reader and the workers exchange data through the 2MB shared SRAM. Workers are both consumers and producers of data because they consume `sensorsBuffer` produced by the PE reader after transposition, and then produce `Module` as a result of processing (Transform into Complex, Compute FFT, Compute Module). We must implement mechanisms to ensure the coherency of exchanged data in `sensorsBuffer` (between PE reader and workers) and `Module` (between workers and PE writer). We use C11 atomic built-ins that bypass the cache on the MPPA K1 architecture.

Control Architecture

The structure is the following. We use two threads on the host: *HOSTsender* and *HOSTreceiver*. We also use 2 threads on the IO cluster core: *IOSenderToHOST* and *IO ReceiverFromHOST*. Finally there are 10 threads in the internal cluster: 8 worker threads (each one on a PE) to compute the algorithms, one PE reader and an one PE writer. Each thread of the internal cluster takes 2 timestamps: one at the beginning and one at the end of its computation. Each worker computes one or several FFTs on 1024 samples and 1 Module.

The thread *HOSTsender* sends packets of 8192 samples (according to the latency experiment) to thread *IOSenderToHOST*. These functions are blocking, so no synchronization is needed between the host processor and the IO cluster.

The thread *IOSenderToHOST* reads input datta and positions its IOWRT0 timestamp. Then, it transmits data to the thread PE reader before positioning its second timestamp ioWRT1. The thread *IO ReceiverFromHOST* reads the data structure coming from the PE Writer thread and sets its 2 timestamps.

8.5.2 Each worker computes 1FFT+1Module of 1024 samples

First we measure the processing time of each worker (see Figure 8.18). This duration is between 214 and 228 microseconds. This means a jitter of 6%. We have the same results from 100 to 1000 loops.

The jitter has two main sources:

- First, it is due to interferences on the shared bus when different PEs are performing access requests to the 2 MB shared-memory. This memory is configured by default in interleaved mode. There is only one bus arbiter which performs a multi-level policy to arbitrate the bus accesses. There are three groups which are arbitrated over three levels: (i) access requests from the 16 PEs in a computing cluster goes through a round-robin arbitration, (ii) then, access from the DMA transmitter, RM and DSU are also subject to a round-robin arbitration, (ii) finally, the DMA receiver uses a fixed-priority policy.
- Second, the execution duration of each worker in Figure 8.18 does not measure exactly the processing duration of a FFT followed by a Module algorithms. But, it also covers the synchronization steps (*wait empty1, wait empty2*) as illustrated in Figure 8.14.

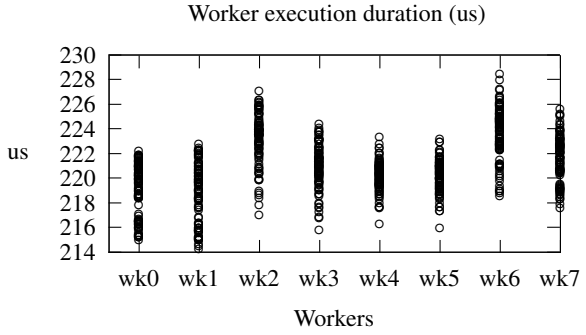


Figure 8.18: Worker Execution Duration after computing 1 FFT and 1 Module in 100 host loops; Host configured in best effort

On Figure 8.19, the x-axis is the timestamp value in microseconds, relative to a major cycle start timestamp. A major cycle is defined by 2 worker loops for convenient periodic display. The first two columns relate to the first worker loop: the beginning and the end of computation respectively. Similarly columns 3 and 4 relate to the second worker loop. The loop in best-effort is around $600\mu s$ with 15% jitter, mainly due to a non real-time HOST.

The duration of workers is given by *column2-column1* or *column4-column3*. Let us take the example of worker0. Its computation starts at the earliest at $600\mu s$ and finishes at 814 μs. This gives a duration of treatment of 214 μs. This value is consistent with Figure 8.18. Between the end of the first packet computation (800 μs) and the beginning of the second one (1200 μs), workers do not make any processing and are waiting, hence no pipelining occurs. The ratio Data Transfert Duration/Processing Duration is equal to $600/214 = 2.8$. To benefit from the computation capabilities offered by the MPPA-256, the processors should compute more algorithms. That is the purpose of the following experiment in which each worker computes 2 FFTs and 1 Module (see Figure 8.20).

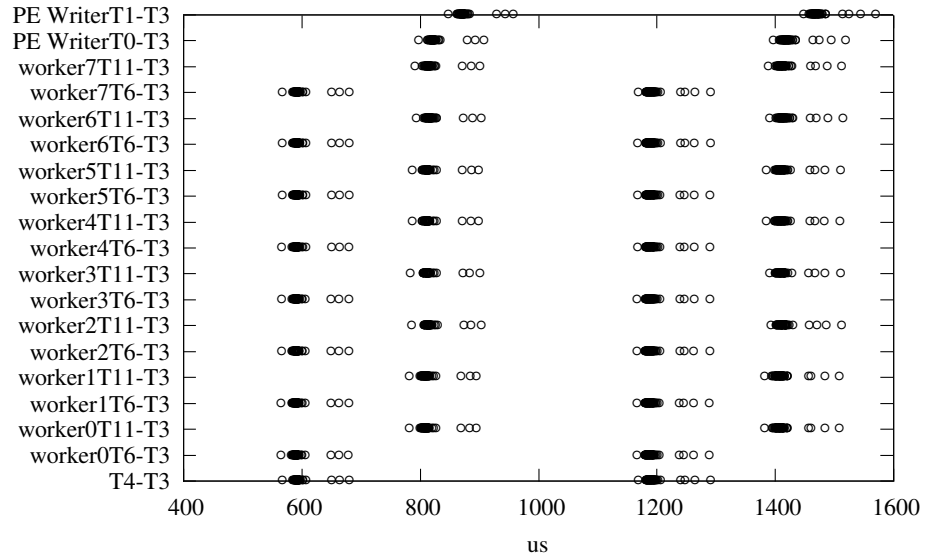


Figure 8.19: 8 Pipelined workers compute 1FFT and 1Module in 100 host loops; Host configured in best effort

8.5.3 Each worker computes 2FFT+1Module of 1024 samples

We repeat 100 times this same experiment. Each of the 8 workers takes between 375 and 400 μs to compute 2 FFTs and 1 Module. This makes a jitter of 6.25%. The HOST sends its samples every 600 μs with a jitter of 15%. The end of treatment of the first packet that was at 814 μs is now at 975 μs. The workers wait only 225 μs (between 975 et 1200 μs) instead of 400 (see Figure 8.21). The ratio Data Transfert Duration/Processing Duration was equal to 2.8 and now becomes 1.6.

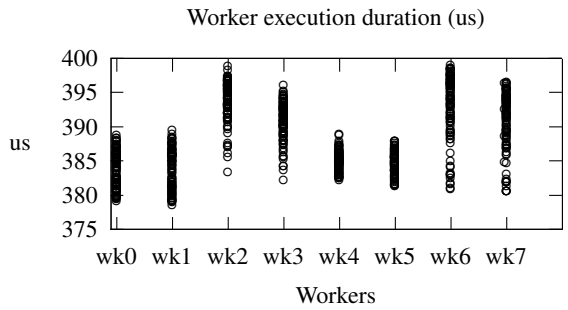


Figure 8.20: worker Execution Duration after computing 2 FFTs and 1 Module in 100 host loops; Host configured in best effort

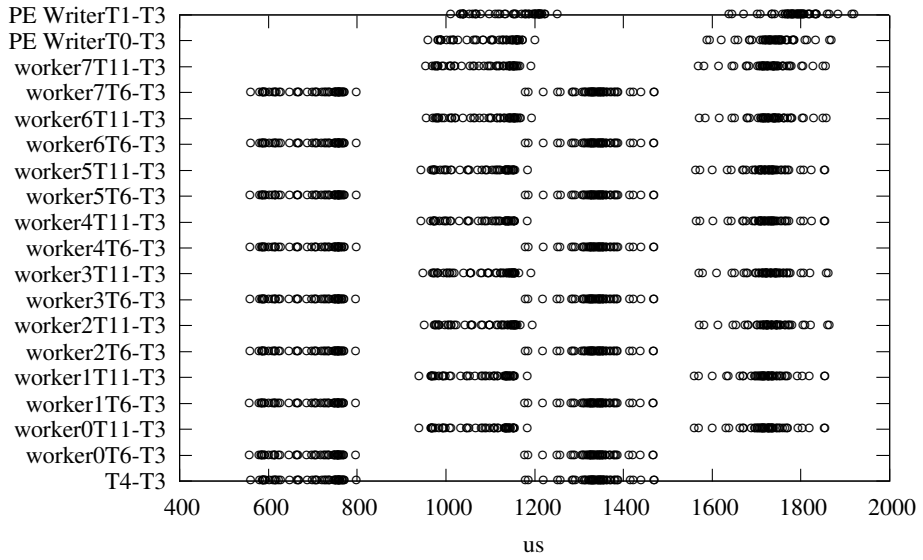


Figure 8.21: 8 Pipelined workers compute 2FFT and 1 Module in 100 host loops; Host configured in best effort

8.5.4 Each worker computes 3FFT+1Module of 1024 samples

We repeat 100 times this same experiment by increasing the workload. Each of the 8 workers takes between 550 and 575 μs to compute 3 FFTs and 1 Module. This makes a jitter of 4.3%. The HOST sends its samples every 600 μs with a jitter of 15%. The end of treatment of the first packet that was at 975 μs is now at 1150 μs. The workers wait only 50 μs (between 1150 et 1200 μs) instead of 400 (see Figure 8.23). The ratio Data Transfert Duration/Processing Duration was equal to 2.4 and now becomes 1.09.

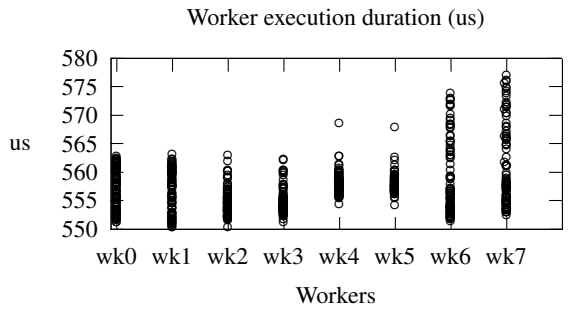


Figure 8.22: worker Execution Duration after computing 3 FFTs and 1 Module in 100 host loops; Host configured in best effort

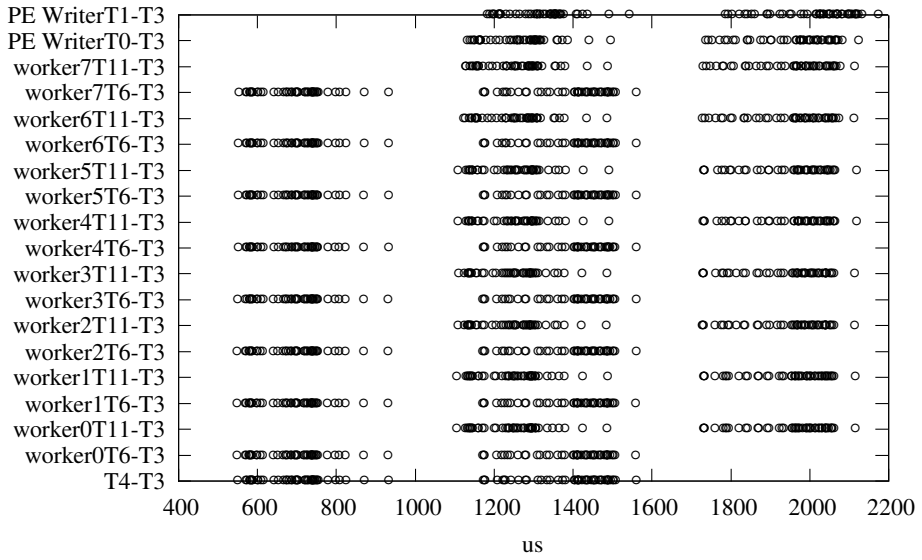


Figure 8.23: 8 Pipelined workers compute 3 FFTs and 1 Module in 100 host loops; Host configured in best effort

8.5.5 MPPA latency jitter

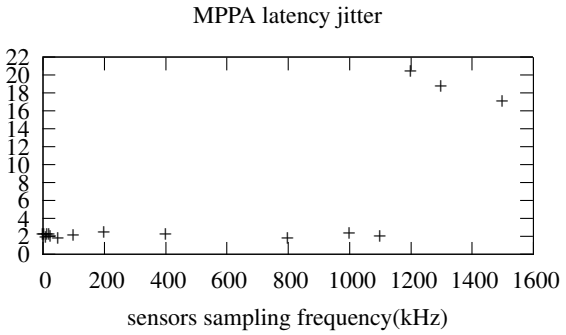


Figure 8.24: MPPA latency jitter measured with various sensor frequencies

The sensors sampling frequency is in the range 1..25 Khz. The period Te_{1024} of sending 1024 samples by the HOST is 600 μs; that corresponds to a frequency Fe_{1024} = $\frac{1024}{Te_{1024}}$ = 1.7 Mhz. With this period a worker is able to compute $600/250 = 2.4$ FFTs. Then we measure the jitter of the MPPA latency using various sensor frenquencies. This jitter is defined by $\frac{(latenceMax - latenceMin)*100}{latenceMax}$.

We notice that it is around 2% at the beginning, before having a peak at 1200Khz. Indeed at low frequency (a long period between two data emissions), the HOST receives the sent data before being able to emit another one. This peak may come from several sources:

- The scheduling of the 2 IO threads made by the operating system, and resulting in a sequential execution of emissions and receptions.
- The internal cluster Resource Manager (RM). The RM also perfoms emissions and receptions in sequence.

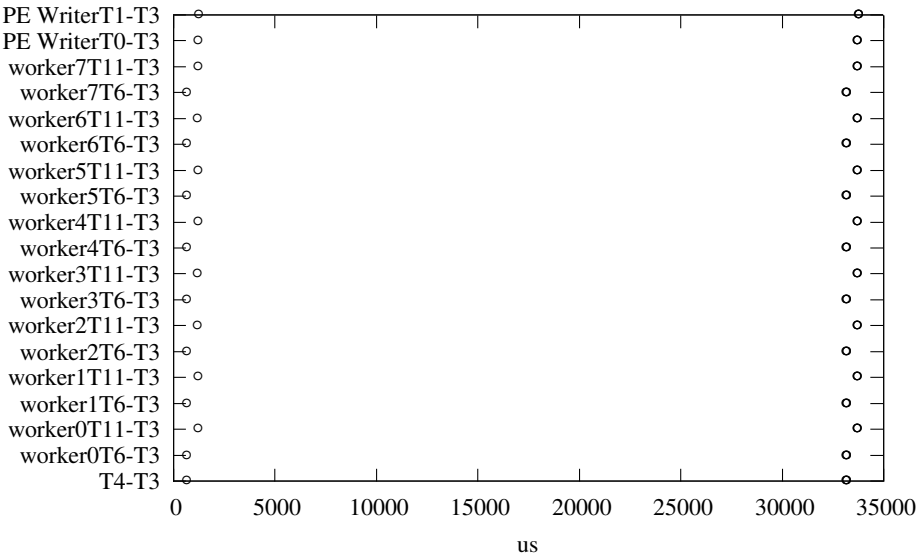


Figure 8.25: Workers pipelined - 3 FFTs and 1 Module - Periodic Host - 100 host loops

Fe_{1024} = $1.7Mhz$ is out of the sensor sampling frequency range. This leads us to configure the HOST with the real sensor sampling frequency.

8.5.6 Pipelined architecture, driven by the bearing frequency

In this last experiment, we choose a particular sensor frequency: $31kHz$ (the bearing sampling frequency). It corresponds to Te_{1024} = $\frac{1024}{31kHz}$ = $33ms$. Each of the 8 workers computes 3 FFTs and 1 Module. The HOST will send to the IO cluster a packet of 8192 samples each Te_{1024}. Then, we represent the pipeline of treatments of each worker on Figure 8.25. It shows that between 2 HOST emissions, each worker can compute for one sensor the equivalent of $Te_{1024}/214$ = 154 FFTs.

8.6 Conclusion

With our choice of mapping, one MPPA cluster of the MPPA processor owns up to 14 workers. Taking into account the parameter ranges fs ≤ $31kHz$ and number of sensors ≤ 256, one MPPA cluster is able to compute a workload of: $(((1024\text{samples}/31kHz)/214\mu s)\times 14\text{workers})/256\text{sensors}$ = 8 1024-plot FFTs, providing the ability to compute several indicators and a comfortable margin for new ones. The MPPA-256 is therefore suitable to perform legacy HMS indicator computation in real-time, and provides extended capability to compute high frequency indicators (MHz sensors), and possibly other avionics functions as soon as other studies demonstrate time and space partitioning capabilities. With its deterministic and predictable (controlled communication and computation jitter) behavior at low operating frequency, this device would tackle the avionics constrained requirements integration/performance/low power/determinism.

Chapter 9

Experiments on Kalray MPPA-256 Processor

This chapter focuses on the implementation of health indicators (temporal and frequency) on the first generation of the Kalray MPPA-256 Processor.

The implementation is done thanks to the results previously described in chapters 5 and 8. As a reminder, Chapter 5 has addressed the transition from the existing global implementation to an incremental version of the health indicators. Indeed, this transition induces a difference but the maintenance decisions remain the same. Chapter 8 first focuses on the input flow by investigating a potential IO bottleneck thanks to the PCie bandwidth and the latency experiments before exploring parallel computation in one internal cluster.

Unlike the previous chapter in which each PE (Processing Element) computes one FFT followed by a Module, here each PE calculates true HMS indicators (OM1, OM2, RMSb and RMSR). The main difference is not only the switch from *realistic* computation (FFT) to *true* health indicators but the tops of the phase are considered in this chapter. Thus, considering the phase sensor requires an important emphasis on the previous steps that allow to compute health indicators: dividing into reference shaft revolutions, dividing into monitored shaft revolutions, interpolating a monitored shaft revolution, calculating an average signal as illustrated in Figure 5.2, page 68.

Finally, we will perform timing measurements on the target in order to verify real-time constraints given by the throughput of input data.

9.1 Real-time constraints

The throughput of vibration samples during an acquisition session depends on the number of sensors, the sampling frequency and the acquisition duration. As a reminder, the HMS function has a few dozen of sensors and the maximum sampling frequency is $31kHz$. Then, the required bandwidth is less than $5\text{MB}/s$. Even if the PCIe GEN3 provides a sufficient bandwidth to transfer once the input flow, the vibration samples will be sent to the MPPA-256 by *chunk* of size VIBRATION_CHUNK_SIZE defined by the latency experiment (refer to Section 8.2.2, page 113). Transferring data by chunk of size VIBRATION_CHUNK_SIZE is mandatory due the limited amount of shared memory in a computing cluster (2MB).

The real-time constraint is set by VIBRATION_CHUNK_SIZE (vcs) acquired at the frequency fs. The transmission of VIBRATION_CHUNK_SIZE through the PCIe, the NoC and its processing must be complete before $\frac{\text{VIBRATION\_CHUNK\_SIZE}}{\text{NUMBER\_ACCEL}\times\text{fs}}$. These different steps are clearly illustrated in Figure 5.2, page 68: the

real-time constraint must be respected between T0 and T1.

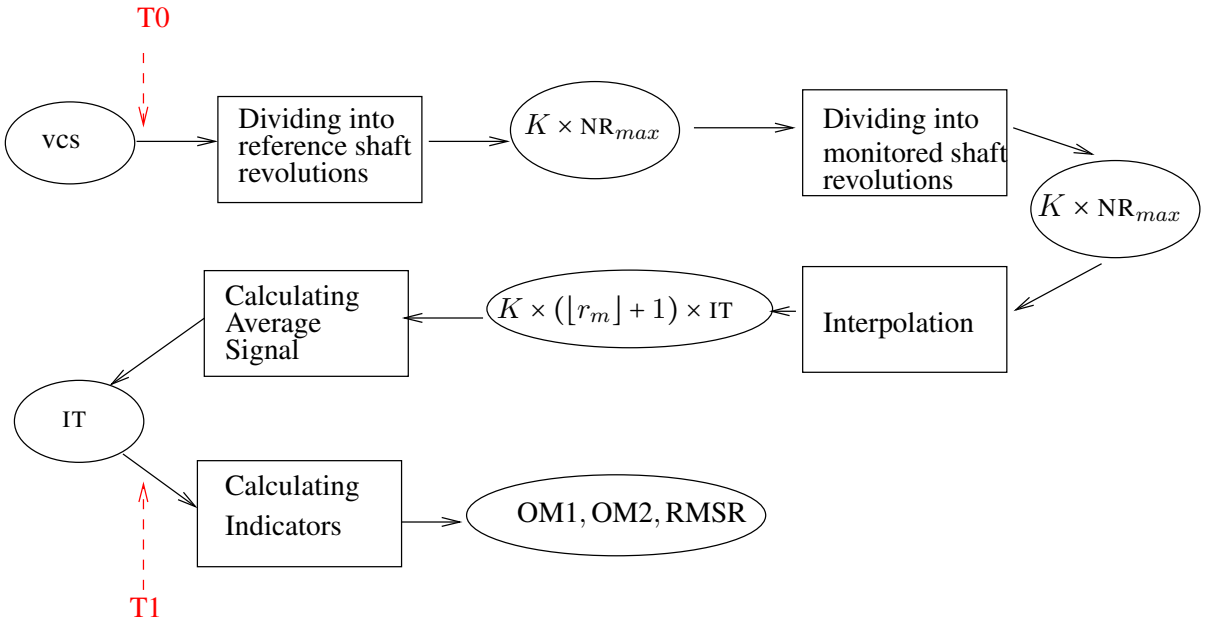


Figure 9.1: Real-time constraint of a worker dataflow from the availability of VIBRATION_CHUNK_SIZE (vcs) to the computation of the indicators. The worker manages a monitored shaft which has a speedup ratio of r_m. T0 indicates the availability of the chunk vcs in the computing cluster and T1 is set once the average signal of size IT is updated.

Figure 9.1 is the worker dataflow. The transformations are represented in squares and the data produced by each transformation in circles. It has two objectives.

(i) First, it illustrates the dataflow of one worker. The worker is dedicated to a monitored shaft which has a speedup ratio r_m. The computation begins when the packet of samples VIBRATION_CHUNK_SIZE (vcs) is available inside the computing cluster memory. Then, this packet is divided into reference shaft revolutions by detecting the tops of the phase sensor and produces

$$\text{NR}_{max} \times K = \frac{\text{SPEED\_RPM\_MAX}}{\text{SPEED\_RPM\_MIN}} \times \text{VIBRATION\_CHUNK\_SIZE}$$

where SPEED_RPM_MIN and SPEED_RPM_MAX are respectively the minimum and the maximum rotational speed of the reference shaft (given in revolutions/minute) during the acquisition session. Then, each reference shaft revolution is split into monitored shaft revolutions. Afterward, each monitored shaft is interpolated. For a given VIBRATION_CHUNK_SIZE, the interpolation step produces $K \times (\lfloor r_m \rfloor + 1) \times \text{IT}$ samples. The interpolation step is followed by the average signal one which produces IT samples. Finally, the indicators are calculated at the *end* of the acquisition session.

(ii) Second, Figure 9.1 also shows the real-time constraint. T0 and T1 are respectively set when VIBRATION_CHUNK_SIZE (vcs) is available inside the computing cluster memory and after the average signal step:

$$T1 - T0 < \frac{\text{VIBRATION\_CHUNK\_SIZE}}{\text{NUMBER\_ACCEL} \times \text{fs}} \; in\,(second)$$

In addition, it is important to note the box '*calculating indicators*' in Figure 9.1) is not considered as a real-time constraint because the indicators are calculated after the acquisition session and only the average signal step is updated for each new VIBRATION_CHUNK_SIZE.

9.2 Example of mapping

We consider a computing cluster among 16 physical available and we take advantage of a temporal and space isolation. We manage to balance computing loads over the cores (processing elements) in order to keep the processing elements busy as much as possible. We want to avoid an unbalance of the workload: keeping a few cores in *idle* while others are usually *busy*. Then, we reconsider the mapping described in the previous chapter for two main reasons: we have illustrated a good load balancing between workers and a huge computing capability which respects real-time constraint. As a reminder, that mapping consists of 2 threads running on the HOST, 2 threads on the IO clusters. In a computing cluster, we have one worker dedicated to one processing element and 2 threads which manage the input/output.

As soon as the workers have different computing loads, we will first assess the real-time constraint discussed above. If the timing constraint is not satisfied, it will be necessary to rethink the mapping.

9.3 Real-time constraints of one monitored shaft

9.3.1 Measuring the execution duration of a reference shaft revolution

For the sake of simplicity, we first focus on the execution duration of each reference shaft revolution. We have only 3 threads running on a computing cluster: one worker and two others threads which manage the input/output flow from/to the computing cluster. The execution duration of each reference shaft revolution is measured according to Figure 9.2: T0 is set once the box '*Dividing into reference shaft revolutions*' produces a chunk of vibration samples corresponding to a reference shaft revolution ant T1 is set after the averaging signal step.

In addition, a monitored shaft may have an *integer* or a *non-integer* speedup ratio (see Section 2.2, page 29). We will show how a non-integer speedup ratio will lead to a burst processing in the following examples.

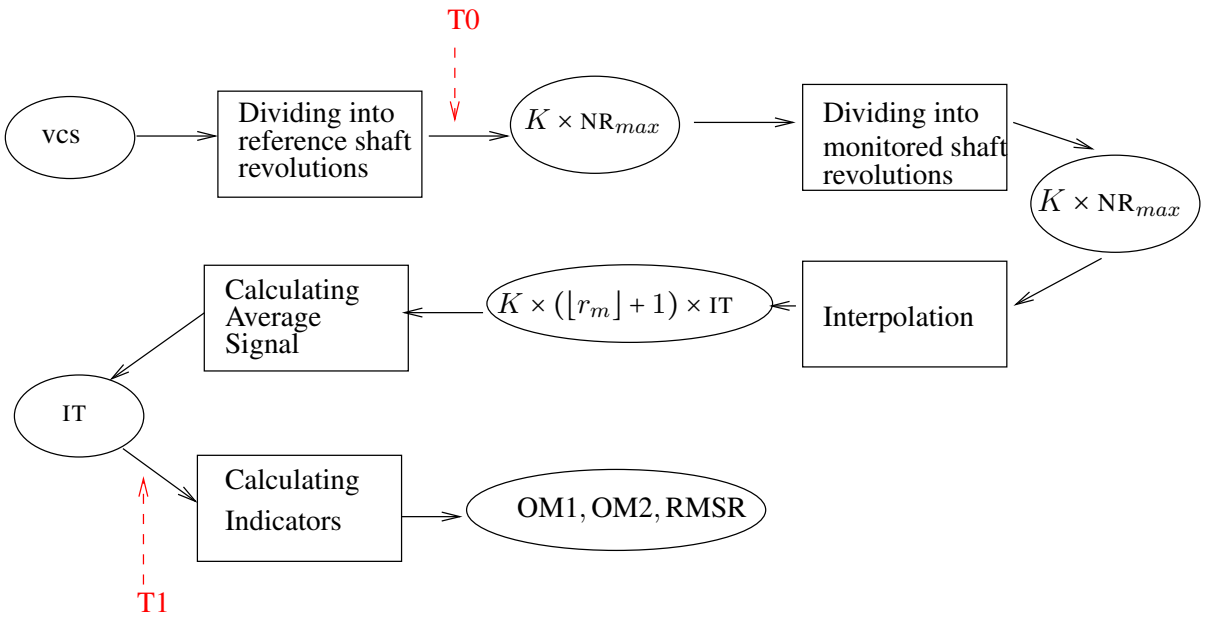


Figure 9.2: Illustrating the timing measurement of each reference shaft revolution.

In the following experiments, we will measure the execution duration of a reference shaft revolution

in 3 cases: with an integer speedup ratio, a non-integer speedup ratio and the maximum speedup ratio.

Case 1: integer speedup ratio

In this example, we consider a monitored shaft which has a speedup ratio equal of 16 with respect to the main rotor. The vibration inputs are acquired at fs = $31250Hz$. Each reference shaft has 1550 samples. Figure 9.3 shows the execution duration of each reference shaft revolution in μs. Each reference shaft has 1550 samples and is computed between 1934 and 1935 μs. This experiment shows that the execution duration of each reference shaft is almost constant: there is no burst processing when considering an integer ratio.

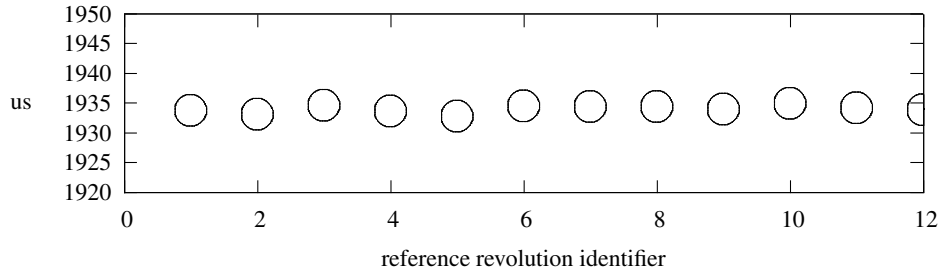


Figure 9.3: Execution duration of each reference shaft revolution with an integer speedup ratio of 16.

Case 2: non-integer speedup ratio

In this experiment, the monitored shaft is the *input drive shaft left* which rotates 16.82 times faster than the main rotor. The vibration data are sampled at fs = $31250Hz$. This experiment addresses the burst processing due to a non-integer ratio. Basically, for each reference shaft revolution, the number of monitored shaft revolutions to process in order to update the average signal is given by the speedup ratio (r_m).

More precisely, the *maximum* number of monitored shaft revolutions for a given reference shaft revolution is equal to r_m if r_m is an integer or $\lfloor r_m \rfloor + 1$ if r_m is a non-integer.

Figure 9.4 shows the theoretical number of revolutions of the *input drive shaft left* for each new reference revolution. The first reference revolution corresponds to 16 complete revolutions of the *input drive shaft left*. The remainder of 0.82 revolution will be completed at the next incoming VIBRATION_CHUNK_SIZE. Then, when the second reference revolution is available, it is composed of 16.82 revolutions but the number of complete revolution to process at this step is 17 due to the previous remainder. At the third step, the new remainder is 0.64 and the number of complete revolutions is also 17 and so on. In general, the number of complete revolutions REV_i of the monitored shaft corresponding to the revolution i of the reference shaft is calculated as follows:

$$\mathrm{REV}_i = \lfloor r_m + \mathrm{REV}_{i-1} - \lfloor \mathrm{REV}_{i-1} \rfloor \rfloor$$

In figure 9.5, we measure the execution duration of each reference shaft revolution. First, we notice that the duration is not constant. Second, this figure has the same shape as the theoretical one. The number of monitored shafts is either 16 or 17 and the corresponding execution time is respectively 1910 and 2030 μs.

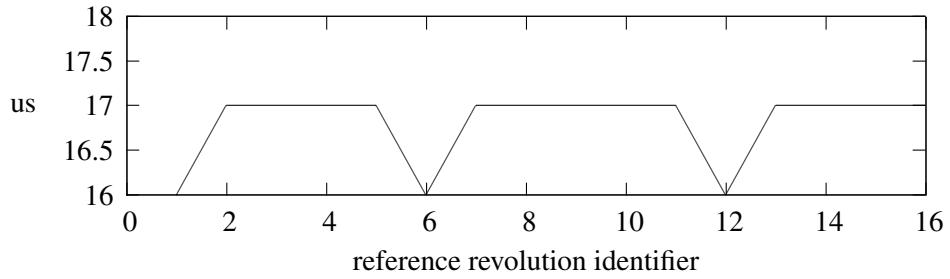


Figure 9.4: Theoretical number of revolutions of the input drive shaft left for each new reference revolution. The speedup ratio is 16.82.

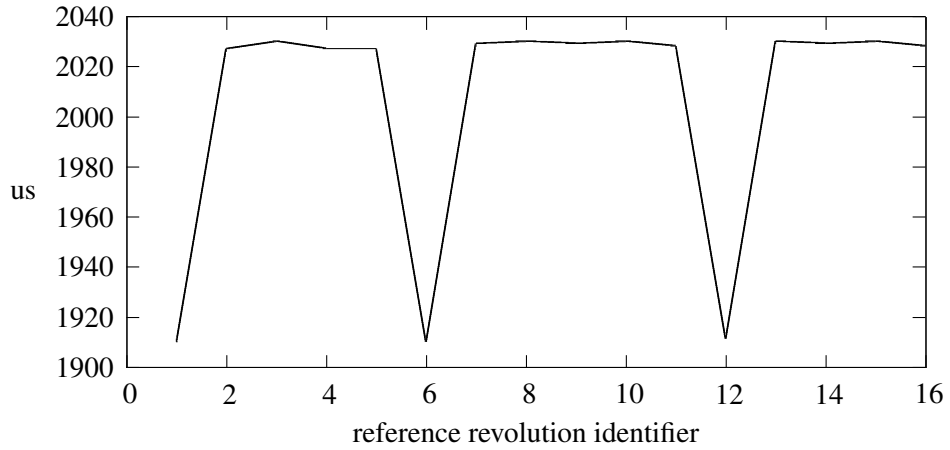


Figure 9.5: Burst processing with a non-integer speedup ratio of 16.82.

Case 3: maximum speedup ratio

The previous experiment shows that the workload scales with the speedup ratio. In this experiment, we tackle the maximum speedup ratio by making a comparison between the execution duration of a reference shaft revolution and the real-time constraint. The maximum speedup ratio is 75. For each reference shaft revolution, we measure the time it takes to update the average signal.

Figure 9.6 shows that it takes around $7300\mu s$ to compute each reference shaft revolution of the monitored which rotates 75 times faster than the reference shaft. Each reference shaft has 1550 samples. Then, the real-time constraint is $1550/\text{fs} = 50ms$. The real-time constraint is met but in a *true* application, the real-time constraint is set according to the input flow. Consequently, in the following experiments, we will consider a constant chunk of samples VIBRATION_CHUNK_SIZE (defined by the latency experiment) to assess the respect of the real-time constraint.

9.3.2 Measuring the execution duration of a packet of constant size vcs

The burst processing in our application has two origins. First, it is caused by a non-integer speedup ratio as illustrated in the previous sections. Second, a burst processing may be generated by a variable number of tops in the transmitted packet (VIBRATION_CHUNK_SIZE) between the acquisition unit and the Kalray-256. As a reminder, this difference is due to the variation of the speed of the rotor (see Section 5.2, page 68).

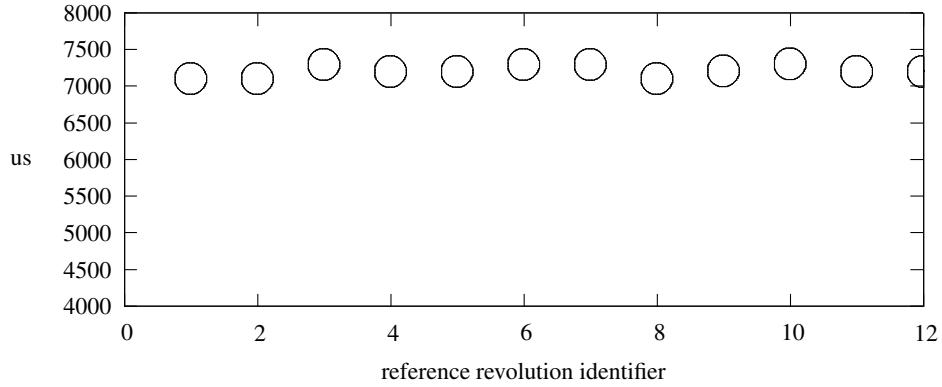


Figure 9.6: Execution duration of each reference shaft revolution with a speedup ratio of 75.

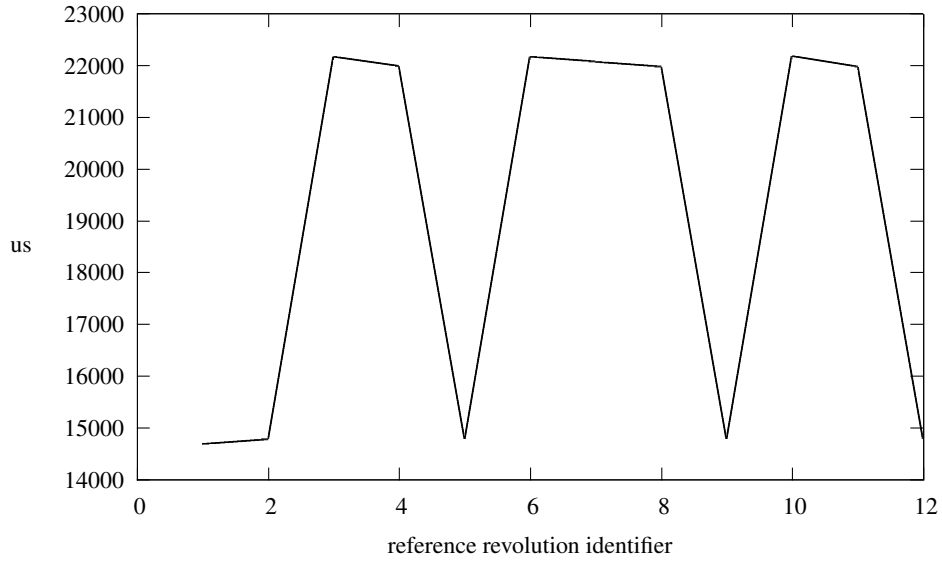


Figure 9.7: Real-time constraint of the maximum speedup ratio equal to 75.

The experiment in Figure 9.7 is the same as the previous one: the monitored shaft, the speedup ratio are the same. The only difference is as follows: we measure the execution duration of each VIBRATION_CHUNK_SIZE instead of measuring the execution duration of each reference shaft revolution. Timestamps T0 and T1 are positioned according to Figure 9.1. We note that there is a burst processing even if the speedup ratio is an integer (r_m = 75). In fact, we choose a VIBRATION_CHUNK_SIZE = 8192 (see the latency experiment in Section 8.2.2, page 113) and each reference shaft revolution has an average of 1550 samples. Then, each packet of VIBRATION_CHUNK_SIZE samples may have 2 or 3 complete revolutions. In Figure 9.7, the first and the second packet of size VIBRATION_CHUNK_SIZE has 2 complete reference revolutions; the execution duration is 15000 μs. The third and the forth packet of size VIBRATION_CHUNK_SIZE has 3 complete reference revolutions and the execution duration is 22000 μs. The real-time constraint equal to $\frac{\text{VIBRATION\_CHUNK\_SIZE}}{\text{fs}} = 132ms$ is satisfied.

9.3.3 Indicators processing duration depending on the interpolation size

We focus on the box *'calculating indicators'* in Figure 9.1. There is no real-time constraint at this step because indicators are processed at the end of the acquisition session. However, the processing of indicators must finish before the next acquisition. Therefore, we need to have an idea about how long it takes. Figure 9.8 depicts the processing duration to compute the indicators depending on the interpolation size IT. Recall, the size of the average signal IT has three possible values: 256, 512 or 1024. It takes 175, 381 and 860μs to compute OM1, OM2, RMSR indicators using respectively an average signal of size 256, 512 and 1024 samples.

In fact, the frequency indicators OM1, OM2, RMSR involved FFT algorithms. In addition, we know that the complexity of a FFT of IT samples is IT $\times$ log(IT). The corresponding theoretical values to process indicators with IT of 512 and 1024 samples are respectively 393 and 875μs. These theoretical values are consistent with our measurements. Finally, the processing of the indicators is less than $1ms$ which is very small compared to $132ms$.

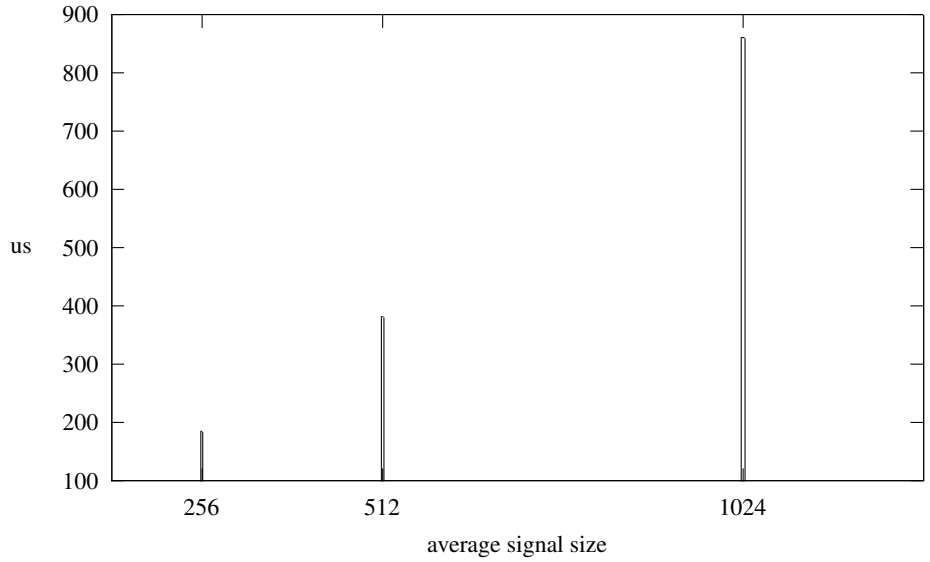


Figure 9.8: Indicators processing duration depending on the interpolation size IT.

9.4 Implementing indicators of several monitored shafts with the same speedup ratios

Here, we consider several monitored shafts with the same speedup ratio. Each worker will manage one monitored shaft which has a speedup ratio of 75. In addition, each worker computes the same indicators. The HOST has a periodic behavior. At each period, it sends to the MPPA a packet of samples of size VIBRATION_CHUNK_SIZE = 8192 acquired at fs = 31500Hz. Then, the period is $130ms$. Figure 9.9 illustrates the pipeline of the internal cluster threads: the PE reader, the 2 workers and the PE writer. PERTDT1 indicates the data availability in the computing cluster. Then, the PE reader dispatches vibration data to the workers. Worker0 and worker1 start their processing in parallel at the same moment. In Figure 9.9, the workers start their processing at Woker0T0=Woker0T1=800 μs and finish at Woker1T0=Woker1T1=16 000μs. Then, the processing duration of each worker is equal to 15200μs. This duration corresponds to the results previously described in Figure 9.7.

Concerning the right part of Figure 9.9, we notice that the workers start their processing $130ms$ later corresponding to the period of the HOST. They finish their processing either at $147000\mu s$ or at 154 $000\mu s$ because each worker can process 2 or 3 complete revolutions of the monitored shaft due to the variation of the speed of the rotor. Again, this observation (the burst processing) is already addressed in Figure 9.7.

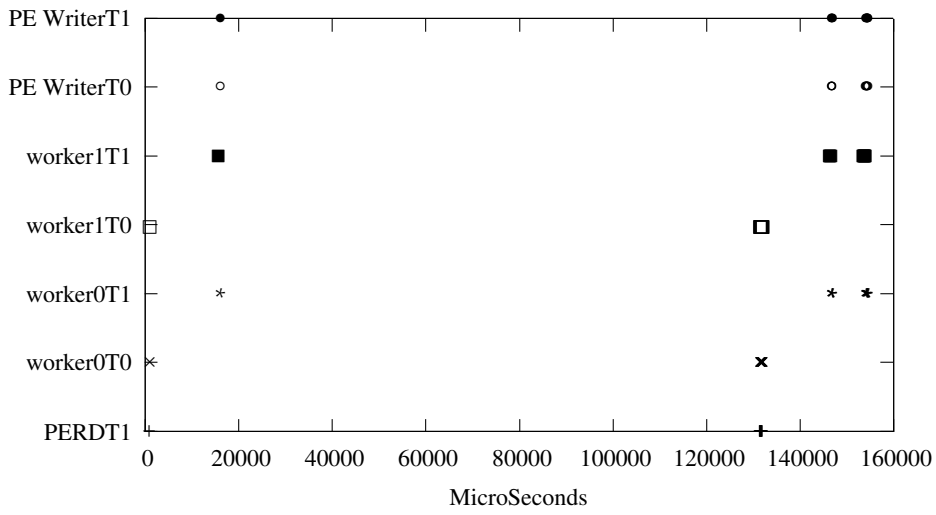


Figure 9.9: 2 Pipelined workers compute indicators. The speedup ratio is equal to 75 and the HOST is periodic. PERTDT1 is the arrival date of the vibration data in the computing cluster. Worker0 and worker1 start their processing at worker0T0 and worker1T0. They finish respectively at worker1T0 and worker1T1. PE WriterT0 and PE WriterT1 are the beginning and the end of the transmission of the results to the IO cluster.

This experiment illustrates four major points: the pipeline of the internal cluster threads (PE reader, PE writer and workers processing are pipelined), the load balancing of the 2 workers (workers have the same workload), the respect of real-time requirement (the processing of the indicators finish before the next period) and finally the burst processing due to the variation of the speed of the rotor.

9.5 Implementing indicators of several monitored shafts with different speedup ratios.

In this section, we discuss about the implementation of several monitored shafts that have *different* speedup ratios. When, the workers compute indicators with different speedup ratios, it means they do not have the same workload. Given an acquisition session, the higher the speedup ratio, the greater the workload.

We have performed experiment to verify if our mapping allows to satisfy the timing constraint when the workload is not equitably distribute over the cores. The following experiment is performed with the maximum speedup (75) and sampling frequency ($31250Hz$). It involved two monitored shafts with different speedup ratio: 16 and 75.

NUMBER_ACCEL	VIBRATION_CHUNK_SIZE	fs	T_{pcie}	T_{NoC}	T_{proc}	Latency	Margin
2	8192	31250Hz	10 us	800 us	22ms	132ms	83%

Table 9.1: Summary of the timing constraint form the sensors to the computing cluster in the worst case

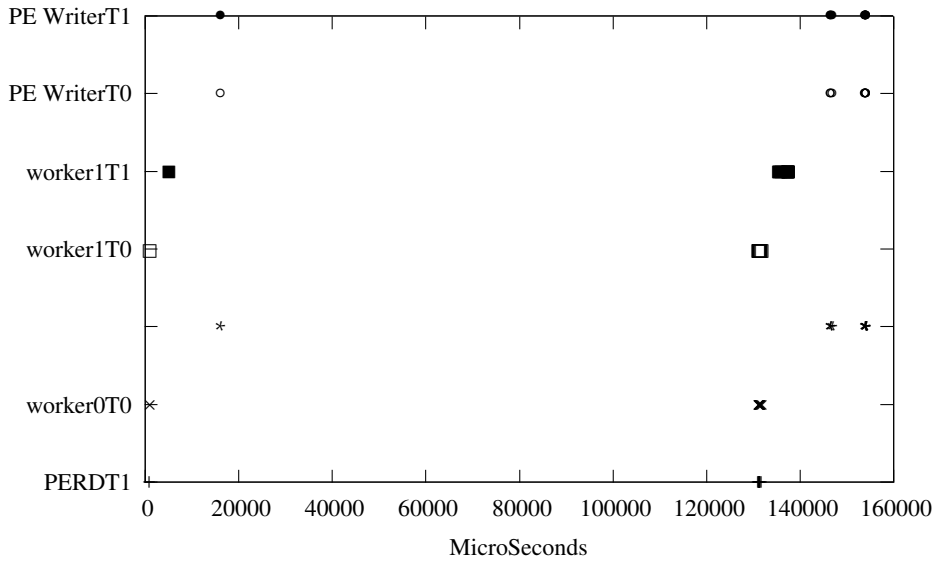


Figure 9.10: 2 Pipelined workers compute indicators. They have a speedup ratio of 16 and 75 and the HOST has a periodic behavior. PERTDT1 is the arrival date of the vibration data in the computing cluster. Worker0 and worker1 start their processing at worker0T0 and worker1T0. They finish respectively at worker1T0 and worker1T1. PE WriterT0 and PE WriterT1 are the beginning and the end of the transmission of the results to the IO cluster.

Like the previous experiment, the data are available inside the internal cluster at 800μs and the workers start their processing at the same date but they do not finish at the same time. In fact, worker0 and worker1 manage respectively the monitored shaft with a speedup ratio equal to 75 and 16. Then, worker0 has more processing load and will finish its processing later. Worker0 and Worker1 finish respectively at 16000 and 5000 ms. The right part of Figure 9.10 shows the burst processing.

Even if the workload is not equitably distribute over the cores, the two workers finish their processing before the second period. We do not need to change our mapping solution because the real-time constraint is met.

Table 9.1 summarizes the timing measurements associated to the data transfer from the sensors to the computation of the health indicators. Each vibration data is sampled at fs= 31 *kHz* and the samples are read per chunk of size 8192. It takes respectively 10 us and 800 us to transfer data trough the PCie and the NoC. The data transfer is insignificant compare to the processing duration which requires 22ms. Finally, each core has a margin of 83% (110ms) to process additional health indicators.

9.6 Conclusion

In this chapter, we have computed real-time health indicators on the processor MPPA-256. We have shown that, there will be a burst processing when the cores are processing indicators. This burst processing is due to either a non-integer speedup ratio or a variation of the speed of the rotor.

Given an acquisition session, our mapping allows to respect the timing constraint given by the input flow. These measurements are also performed in the worst case: the maximum speedup and frequency. In addition, we note that the different speedup ratio will lead to an unbalance workload over the cores. However, even if the cores have different workload, our mapping guarantees the respect of the timing constraint. We have also addressed a parallel processing and a pipeline architecture.

In fact, we have not implemented the bearing indicators. The bearing indicators allow to detect the defect frequencies of the four parts of the bearing: the inner race, the outer race, the rolling elements and the cage. However, these indicators are based on the same principle as the indicators we cover in this chapter. The bearing indicators involved the same algorithms like FFT and reverse FFT. The comfortable margin provided by the processor allows to implement these indicators without an impact on the respect of the timing constraint.

Finally, it is important to note that we have shown the respect of the timing constraint by performing few measurements. However, based on the determinism of the processor (see Chapter 8, page 107), we assume that these measurements are close to the worst case execution time. We have to perform statistic measurements to assess the respect of the real-time constraints.

Part III

Related Work and Conclusions

Chapter 10

Related Work

The early detection of mechanical failures is crucial in the avionics domain. This early detection allows to avoid the progress of the defect that could lead to the fail of the system. For instance, according to [72], an amendment proposed by EASA in 2010 covered mainly vibration health monitoring of the *drive train*. The drive train is responsible for transferring power from the engines to the rotors. It is a critical part due to the high variability of the dynamic loads acting on the drive train. The load of the helicopter varies during the flight and depends also on the pilot operations. According to [73], a yaw angle deviation of 30° and 45° produces respectively a dynamic load variation of 1.15 and 1.41.

The Health Monitoring System (HMS) has been introduced in the helicopter's industry in the last two decades and has been expanded in various areas such as fixed-wing aircraft, drones and business jets (refer to [74]). It allows to monitor the health of vibrating components. Because the defect is not directly observable, it is mandatory to measure physical signals which are linked to the state of the machine. The sensors attached to the rotating components measure the vibration signatures of the rotating components. The HMS function is composed of two types of sensors: accelerometers and phase sensors. It includes also contextual parameters of the helicopter such as air speed, pitch altitude, roll altitude, the engine states, etc. The contextual parameters are usually provided by sensors connected to the avionic navigation system rather than to the HMS.

10.1 Towards Condition-based maintenance

The helicopter components are usually replaced at fixed intervals. The *systematic maintenance* (see Section 1.1.1, page 13) is carried out at a constant time period according to a schedule. There is no intervention before the deadline determined in advance. This has two major disadvantages. First, unnecessary components may be replaced which will generate additional costs. Second, a defective component may not be replaced which may cause accidents and therefore safety issues.

However, the HMS function is moving towards a *condition-based maintenance* which focuses on the health of the helicopter rather than a scheduled maintenance. The condition-based maintenance avoids unnecessary maintenance. Thus, the HMS function reduces the *maintenance costs*. Another benefit of the HMS function is the *flight safety*. According to [74], the HMS installed on the Boeing Chinook has decreased the average maintenance time for rotor balancing from around 10 hours to around 3 hours. The trend of the health indicators can help to anticipate a mechanical defect. According to [74], the analysis of vibration data after the accident of the Boeing Apache helicopter from which an operator crew died, has identified three additional helicopters that would have eventually crashed.

10.2 HMS for other manufacturers

10.2.1 Sikorsky

The helicopter S-92 from Sikorsky provides since March, 2017, the capability to transmit raw data in real-time to the Louisina operations center. Then, the ground support team can process, visualize, analyze health indicators and provide helpful information to the on-board operators. Sikorsky's S-92 has 140 sensors capable of tracking more than 5000 parameters. The data transmission is performed by a *satellite communication* that supports voice, analog video and digital aircraft system information.

In the former version of the HMS, vibration data was acquired during the flight and the health indicators were processed when the helicopter was on-ground. However, the current HMS offers the capability to compute health indicators at the ground station while the helicopter is flying. Such a capability increases the availability of the helicopter. The on-ground operators analyze the trend of the indicators, take operational and maintenance decisions. For example, a decision may be an *"immediate landing is necessary"*.

According to [75], Sikorsky says that this is the first satellite that transmits real-time flight data monitoring system for civilian helicopters. The new HMS brings a significant progress in technology that avoids the radio communication dependencies between the operators and the pilot (refer to [76]). In addition, the data transmission in real-time addresses the early failure detection which could be life-saving. According to [74], the ultimate goal of the current HMS is to provide an ample notice of impending issues and to prevent operators of problems of 60 to 100 flight hours in advance.

However, to the best of our knowledge, we do not find any article that tackles the real-time constraints of the new HMS proposed by Sikorsky. A tight timing constraint is crucial to detect rapid change of the trend of the indicators. Finally, Sikorsky is also working on to move from condition-based maintenance to predictive maintenance. A predictive maintenance allows to determine when *corrective* maintenance should be performed. The predictive maintenance schedules are determined by analytic algorithms (see Section 10.3 for additional details about predictive maintenance in the industry).

10.2.2 Aviation Industry Corporation of China (AVIC)

Aviation Industry Corporation of China (AVIC) has proposed a method to evaluate the health of the helicopter main gearbox. This choice is motivated by numerous accidents caused by the main gearbox. According to [12], 10% flight helicopter accidents are caused by the main gearbox. The health method is based first on the fusion between flight parameters and indicators. Second, it uses a classification model to separate the healthy and faulty groups. The computation is performed on-ground.

It is common to monitor the health of the helicopter by focusing only on vibration data. But, it is very difficult to assess accurately the health of the helicopter based only on the vibration data because the variation of the operation conditions are changing during the flight. As a reminder (refer to Section 2.7, page 30), the vibration measured on board are not only caused by the mechanical defect. They take into account the pilot operations, the natural vibration caused by the structure of the helicopter itself and the vibration coming from others components. Based on this observation, AVIC proposes a new condition indicator based on the effect of the flight parameters. Their study can be grouped into two main ideas.

The first step is devoted to the selection of the flight parameters that should be related to the change of the health indicators. The flight parameters include the rotor speed, indicator air speed, atmospheric pressure, altitude, yaw angle, pitch angle, roll angle, engine states, etc. The relation between the health indicators and the flight parameters is calculated according to the following formula:

$$COV(X_i, Y) = \sum_{j=1}^{N} \frac{(X_{ij} - \bar{X}_i)(Y_j - \bar{Y})}{N} \tag{10.1}$$

X_i are flight parameters, Y is a given health indicator and N the number of indicators.

A threshold of correlation is set and the most relevant flight parameters are selected according to that threshold. A multiple regression model is used to evaluate if there is a statically significant relation between a set of variables (see [77]). In this study, the variables are the health indicators and the flight parameters. A multiple regression is almost the same as a linear regression except for the number of variables.

After analyzing the data, they notice that the health indicators are affected by the *engine torque, rotor speed, indicator air speed and the altitude*.

This result is in accordance with our experimental results. As a reminder, in Section 4.10, page 63, we have discussed the limit of the convergence criterion. We have considered 2 acquisition files which satisfy the convergence criterion (refer to Paragraph 3.6, page 43). The only difference between these two files are the conditions operation. One is acquired when the speed of the rotor is almost constant and the other during a regular acceleration phase. We note that when the speed of the rotor is constant, the OM1 indicator is also constant. On the other hand, it increases with respect to the speed variation. Our experiment shows that there is a positive correlation between the OM1 indicator and the rotor speed. We state that the flight parameters should be taken into account to build a rigorous model of the HMS.

The second idea of the paper is a classification model based on the *Mahalanobis-Taguchi Distance (MD)*. The model has as inputs the relevant flight parameters and the health indicators. The indicators are classified into two groups depending on the MD distance. The two groups can be distinguished from each other by using the MD distance. According to [78], the MD is a measure of the distance used to determine the similarity of a known set of values (normal cluster) to that of an unknown set of values (abnormal cluster). It is based on the correlation between variables and is different from *Euclidean distance* because it addresses the correlation and the distribution of the data.

After calculating the MD of each indicator, a threshold value is set which allows to set the boundary value of the two classes (normal and abnormal). Their model is tested separately due to the lack of real gear faulty data in flight. It is first verified by data acquired during the flight and other by the faulty data of the main gearbox. The model proves to be efficient to evaluate the health status of the pinion of the main gearbox of helicopter.

The MD should be explored in the scope of the HMS. It would help to move from a threshold based on experience to a classification model.

10.3 Towards using predictive maintenance in the industry

Predictive maintenance is defined as a set of techniques that allow to determine the condition of in-service equipment in order to predict when maintenance should be performed (see [79]). Like a condition-based maintenance, predictive maintenance is also based on the performance of the machine. Thus, it aims at reducing the maintenance costs by eliminating unnecessary systematic maintenance. In contrast to condition-based maintenance, preventive maintenance is performed at scheduled point in time. It uses statistical approach to determine the suitable moment to perform maintenance in the future.

Predictive maintenance evaluates the performance of the equipment by performing periodic or continuous monitoring. It is usually performed when the equipment is in normal service in order to avoid disruption of normal system operations. To evaluate the properties of the equipment, predictive maintenance uses non-destructive technologies such as acoustic, infrared, vibration analysis etc.

According to [80], industries spend approximately 80% of their time reacting to issues rather than proactively preventing them. Companies that are involved in predictive maintenance program can expect to have significant improvements in terms of reliability and cost efficiency, such as: 10x Return on Investment (ROI), 25-30% reduction in maintenance costs, 70-75% elimination of breakdowns, 35-45% reduction in downtime, 20-25% increase in production.

Predictive maintenance is more complicated and needs more effort to deploy compared to systematic maintenance. It requires also a data science expertise and employees must be trained to interpret the results. That is the reason why companies like *Mathworks* have developed a framework to facilitate the development of machine learning algorithms.

10.3.1 How Machine Learning works?

Machine learning is a set of techniques that teaches computers to do what comes naturally to human and animals: getting better at some task through practice (refer to [81]). Machine learning algorithms use computational methods to "learn" information directly from data without relying on a predetermined equation as a model. The algorithms adaptively improve their performance as the number of samples available for learning increases. There are three different types of machine learning algorithms:

Supervised Learning

A supervised machine learning algorithm takes a training set of examples with known responses to the data (output) and builds a model that generalizes to respond correctly to new inputs. *Classification* and *regression* are examples of supervised machine learning techniques.

Unsupervised Learning

In unsupervised learning, correct responses are not given. The algorithm finds hidden patterns, similarities between inputs without labeled responses. The inputs which are something in common are grouped into the same class. *Clustering* is the most common unsupervised machine learning technique.

Reinforcement Learning

According to [81], this is somewhere between supervised and unsupervised learning. The algorithm gets told when the answer is wrong, but does not get told how to correct it. It has to explore and try out different possibilities until it works out how to get the right answer.

10.3.2 Predictive maintenance with MATLAB

MATLAB machine learning toolbox:

The idea behind the MATLAB machine learning toolbox is to make easy, convenient the use of big data and accessible to engineers without having an expertise in statistics. According to [82], *"There's no need to learn big data programming or out-of-memory techniques – simply use the same code and syntax you're already used to."* The big data workflow is depicted in Figure 10.1.

- Access Big Data: This step consists of accessing data that are too large to fit in local memory. A *datastore* is used to import files such as tabular data, images, databases in small amount that do fit in memory for analysis.

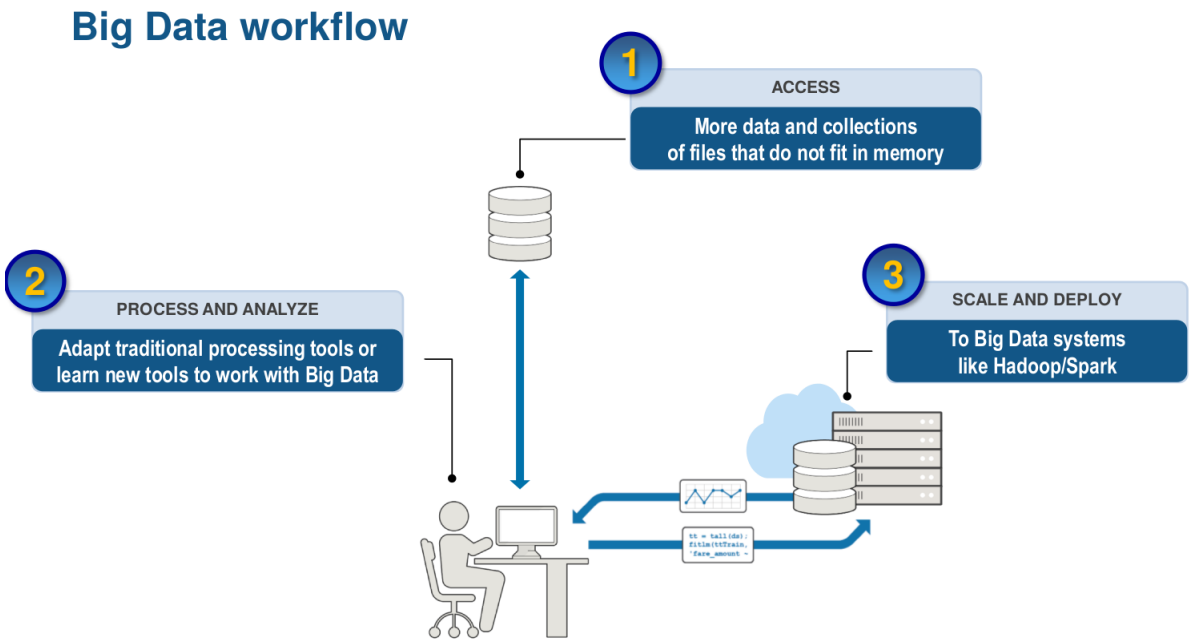


Figure 10.1: Big data workflow with MATLAB, extracted from [82].

- Process and Analyze data: MATLAB provides additional features to process large amount of data such as *parallel-for loops* are used to speed-up for loops with independent iterations. Again, there is no need to change the code: just replace *for* with *parfor*. In addition, it supports also the *MapReduce* function. The map function processes data in chunk and the reduce function calculates the output for each key (refer to [82]). After the processing step, the experts select the relevant parameters, features to consider in order to build a predictive model.
- Scale and Deploy: MATLAB offers also the capability to scale applications from single workstation to compute clusters. After creating a predictive model on the local workstation then and run the applications where your data lives using *Hadoop* (refer to [83]) and *Spark* (see [84]).

Industrial use case using MATLAB machine learning toolbox:

Safran used the MATLAB machine learning toolbox from Mathworks to develop predictive maintenance algorithms for their Turbofan engines. This is beneficial for airlines which invest substantial resources in maintenance, repair and overhaul (MRO). Maintenance delays are key important issues for passengers as well as airlines. According to Airlines for America (see [85]), the cost of maintenance delay is 11.63$ per minute.

The use case from Safran aims at improving the aircraft availability (on time departure and arrivals), reduce engine out-of-service time and the maintenance costs.

The sensor data are gathered from 100 engines of the same model over 360 flights. The use case study is developed as follows to predict and fix failures before they arise: pre-process and analyze historical sensor data, design a model to predict when failures will occur, deploy the model to run in real-time, predict failures in real-time. It can be very overwhelming to choose the right machine learning algorithm because there are dozens of algorithms and each one is based on different learning techniques. There is no common method to choose the best algorithm for a given problem. Even experienced data scientists have to try different algorithms. MATLAB provides an intuitive user-interface that allows to compare

different machine learning techniques very quickly and to choose the one that gives the best prediction score.

10.3.3 Amiral Technologies

Resulting from several years of research at the National Center of Scientific Research (CNRS) in Artificial Intelligence, Automation and Control, *Amiral Technologies* aims at revolutionizing the data mining for industrial sectors (see [86]).

The main innovation of Amiral Technologies is an Automatic Generation algorithm of discriminating characteristics from time series whatever their nature. Experts usually intervene during the analysis step to select the relevant features *before* building the predictive model. However, using the feature generation from Amiral Technologies, the intervention of the experts is needed for interpretation *after* the elaboration of the model. Their predictive models cover the health monitoring domain like fault detection and End-of-Life prediction (refer to [87]).

Estimating Remaining Useful Life (RUL) of Power Contactors

This example aims at estimating the remaining useful life of the power contactors. The question to answer is the following: Is there any property of the discharge current signal that increases monotonically with the contactor state-of-health? Two predictive models are proposed: one based on the original features and the second one using the automatic feature generation. These two models are compared based on the error of prediction. The error of prediction can goes up to 40% when considering only the original features and 14% thanks to the generation of features.

10.4 Real-time implementation case studies using the MPPA-256 many-processor

The next sections will be devoted to the design of applications running on the manycore processor MPPA-256. The first study aims at implementing a real-time hyperspectral SVM (Support Vector Machine) classifier. The second one focuses on porting hyperspectral image processing on a manycore platform by performing optimization to fulfill real-time constraints.

10.4.1 SVM-based real-time hyperspectral image classifier on a manycore architecture

The article entitled "*SVM-based real-time hyper-spectral image classifier on a manycore architecture*" was published on August 2017 in the journal "*Journal of Systems Architecture*". This publication (refer to [88]) is a collaborative project between two spanish universities (UPM) and (ULPGC).

The purpose of this study is to provide real-time detection capability of brain cancer during neurosurgical procedures at hospital.

The images are sent to the manycore processor (*MPPA-256*). A supervised machine learning algorithm (*SVM*: Support Vector Machine) [89] is used to classify images into 4 families (healthy tissue, tumor tissue, hypervascularized tissue and background). The images are scanned between 1 and 2 minutes. The end-to-end processing duration should not exceed 1 minute.

Database

Hyper-spectral imaging is used in this study to detect tumors in the brain. This technology allows to reconstruct a scene with spectral bands. Generally, it is possible to differentiate the different components

of an image and their proportions after analyzing the spectral response.

The images come from the *HELICoid* project (see [90]) and they are acquired during neurosurgical procedures in Spain and England with 2 hyper-spectral sensors. In addition, their sizes are respectively **60.6MB** and **129.1MB**. They are large due to their high resolution information: 124 033 pixels for the 1st sensor and 264 408 pixels for the second one.

System Architecture

The System architecture is divided into three levels: a host module, an I/O subsystem and 16 clusters. The host is connected to the I/O cluster through a PCIe bus and contains the largest memory of the global system. The communication between the host and the computing clusters is managed by the I/O. It is connected to the computing cluster through the NoC. The I/O and computing clusters have respectively a memory block of 4GB and 2MB. There are one thread running on the host, one thread on the IO cluster and up to 16 threads on the internal cluster.

Internal cluster processing

The internal cluster memory of the MPPA-256 is 2MB. In addition, an amount of this memory is blocked for the operating system and another part is dedicated to the application. So, the remaining part for the application is around 1.5MB. As a result, the images will be split into several blocks in the I/O cluster before being transferred to the internal cluster.

Each image contains 128 spectral bands. The size of each band is $\frac{1.5\text{MB}}{128}$ = 12KB. In addition, each pixel is a float (4 bytes), the number of pixels per band is $\frac{12\text{KB}}{4}$ = 3072. Then, each spectral band can contain at most 2048 pixels which is the maximum power of 2 within the possibilities. Consequently, the IO cluster will send to the internal cluster an image of size 1MB. Finally, the cluster processing is parallelized and each core will process 128 pixels of the block image.

Communication

The Host sends the entire image to the IO cluster. However, since the internal cluster has only 1.5MB available, each image will be divided into chunks of 2048 pixels. A blocking communication mechanism (*channel*) is used between IO and internal cluster. A double buffer technique is also implemented to parallelize data emission and processing.

Processing time with the number of cores

In this experiment, the processing time to classify each block of image is measured from 1 to 16 cores and the associated speedup is calculated. In both cases (image of 60.6MB and 129.1MB), the SVM is processed in less than 4ms. Which means that each pixel is calculated in $1,6\mu s$ and $1,8\mu s$ respectively. The maximum speedup when considering 16 cores running in parallel is equal to 14.

Processing versus Transmission

The transmission of the first entire image between the host and the IO cluster is $14.44ms$; it takes $353.3ms$ to transfer the image from the IO to the internal cluster.

In addition, experiments shows that the transfer of an image block from the IO to the internal cluster takes $5.79ms$ while its classification duration is $3.25ms$. In others words, the core is in *idle* mode during $2.5ms$. During the idle time, the cores do not do any computation and are waiting for data coming from

the IO cluster. This experiment shows that the blocking factor is the IO cluster which is a bottleneck. Thus, the transmission of IO data to internal cluster is improved by using the double buffer technique.

Double buffer technique

The double buffer technique is implemented using 2 input and output buffers to communicate between the IO and the internal cluster. The double buffer allows to parallelize the transmission and the processing of each chunk of image.

Discussion

Experiments have illustrated that the blocking point remains the data transfer from the host to the internal cluster. Indeed, when a simple buffer is used, the global system spends more time on communication than on processing. Our experiments in the scope of the HMS function have also demonstrated the same observation (see Chapter 8, page 107). This was our first conclusion when we started to implement algorithms using the manycore MPPA-256 processor. The ratio between the data transfer and the processing is equal to 2.8 (refer to our preliminary experiments, Section 8.5.2, page 126) when each core computes one FFT followed by a Module. This means that, we need to increase the processing load to keep the cores busy as much as possible and to use efficiently the processing capabilities offered by the processor. Finally, even if the second generation of the MPPA-256 processor (Bostan, see [91]) is used in this study, we note that the IO bottleneck issue is still present like in the Andey version.

In addition, a double buffer optimization has been implemented to parallelize the image transmission and the cluster processing. By doing so, the IO bottleneck has been located. In our case study, we have also designed a pipeline software architecture in the computing cluster to improve the throughput. The pipeline works like a double buffer because we have a dedicated thread which reads the input flow from the IO while the cores compute indicators (refer to Section 8.3, page 115).

We cannot compare the processing duration in the internal cluster, even less the different speedups because the algorithms computed are not the same: Support Vector Machine and Health indicators. But, the duration of the data transmission over the NoC given by both approaches are the same. It takes $353.3ms$ to transfer an entire image of size 60.6MB which corresponds to a transfer rate of **171.52 MB per second**. Concerning the HMS function, the MPPA-256 latency experiment in Figure 8.7, page 115 shows that it takes approximately $1500\mu s$ to transfer 128KB in a round-trip between the IO and computing cluster, which gives a transfer rate of **174.76 MB per second**.

Finally, the software architecture proposed in this study can be improved by implementing temporal parallelism on the host and IO cluster. Multitasking on the host and IO will improve the throughput and the host will no longer have to wait the end of processing of the packet before sending another one.

10.4.2 Porting a PCA-based hyper-spectral image dimensionality reduction algorithm for brain cancer detection on a manycore architecture

This paper (see [17]) presents an intrinsic analysis of a PCA (Principal Component Analysis) algorithm (see [92]) and its adaptation to a manycore processor (MPPA-256) by exploiting the parallelization opportunities offered by the manycore processor. The PCA algorithm is applied to hyper-spectral images in order to reduce the image dimensions before to start the processing itself. In addition, the impact of the communication between the Host and the internal cluster is also evaluated. Finally, a comparative study is performed with other PCA implementations on the state of the art which target a FPGA *virtex7* from Xilinx (refer to [93]) and *Intel-i7* processor.

PCA analysis

The hyper-spectral images are composed of several spectral bands that are usually correlated: it contains a redundant information. The number of bands is 128. This redundancy can be reduced using a PCA analysis. The PCA converts the original image into smaller dimensions by removing the dependencies between the different bands. It consists of calculating a covariance matrix of the original image and then extracting the eigenvectors (see [94]) from the covariance matrix. Finally, the original image is projecting on the set of eigenvectors. The different stages of a PCA algorithm are listed below (Y is the original image composed by N pixels per M spectral bands):

(i) *X = Remove the average of each spectral band of Y.* The first step centers the original image by removing the average of each spectral band.

(ii) *Covariance matrix* $C = X^T \times X$. The second step calculates the covariance matrix associated with the original centered image. The dimension of the covariance matrix is $M \times M$.

(iii) *E = Eigenvector decomposition of C.* The eigenvector of the covariance matrix is extracted at this step.

(iv) *Projection of the original image to* $E : Q = Y \times E$

(v) *Reduction from Q to P principal Components.*

Sequential implementation

The sequential version of the PCA is implemented by considering 1 cluster and 1 core. Among the five steps of the PCA, the covariance matrix is more time consuming. Its processing represents **92 %** of the global execution time which is equal to 44 305.2ms for the first image and 87 321.8ms for the second one. The sequential version does not have performance required to meet the real-time constraint.

One level of parallelism: 16 clusters, 1 core

The PCA algorithm is distributed over 16 computing clusters with only 1 core used per cluster. The overall execution time has decreased but very slightly. The equivalent speedup is **1.5 %** using 16 clusters in parallel. The speedup is too low because ideally, it should go up to 16. This limitation is due to the broadcast of data between the IO and 16 computing clusters.

Two levels of parallelism: 16 clusters, 16 cores

The overall architecture is considered. The speedup achieved is negligible: it accelerates only the processing by a factor of 1.1. The reason that explains that result is that most of the time is dedicated to the communication over the NoC. Finally, additional experiments about the data transmission between the IO and computing clusters have shown that: when the PCA is completely parallelized, more than **99%** of the covariance is dedicated to the communication.

Communication improvements

The bottleneck of the system is the data transmission on the NoC. To overcome this issue, the computation of the covariance matrix which spends 99% on communication is moved to the HOST where the available memory is more than 10GB. This improvement allows to achieve a speedup of **20**.

Comparison with others targets

The results are compared with those in the state of the art which target the Xilinx virtex7 and Intel-i7 processor. Because these experiments use images of different sizes, the comparison will not be done in terms of processing time but in processing time per MB. The processing time per MB are respectively 9.8, 29.8 and 36.2ms for the Intel-i7 processor, FPGA and MPPA-256. We note that the memory limitation is a crucial factor because the Intel i-7 processor has lower memory constraint. This difference can also be explained by the difference in processor frequency (3.6GHz for Intel i-7 and 600MHz for the Bostan version of MPPA-256).

However, it would not be very judicious to compare processors by considering only the computing capacities. The power consumption is also an important factor to consider. The Intel-i7 processor can not compete with FPGA and MPPA-256 processor in terms of power consumption. It consumes on average **84W** while the FPGA (**30-40W**) and MPPA-256 (**5W**).

Discussion

In analogy, each accelerometer or contextual parameter is a spectral band. The number of bands is below 128 in each acquisition file. However, as discussed in Chapter 2, page 20, all accelerometers and contextual parameters are not in the same acquisition file. We would use PCA analysis if vibration data and contextual parameters are gathered in the same file.

In addition, the number of bands is not the only factor that justifies the need to use PCA. It is also necessary that the information of the different bands is significantly redundant. This is what should be studied in the context of vibrations: is there a correlation between the vibrations measured by the accelerometers on the main rotor and main gearbox?

Finally, the hint consisting of moving a computational load which involves a lot of NoC data transmission to the Host is not convenient in our application. Since, the on-board HMS function is composed of sensors connected to an acquisition card and a processing device. We use only the host to mimic the sensors.

10.5 Conclusion

In this chapter, we have positioned the current technologies in use for early failure detection. Avionic companies are exploring various technologies in maintenance domain. For instance, Sikorsky is moving from on-ground condition-based maintenance to an on-ground real-time failure detection with a vibration data transmission based on satellite communication. In addition, helicopter manufacturers like AVIC are dealing with an on-ground predictive maintenance. They use a classification technique to separate normal and abnormal classes. In this chapter, we have also positioned the new framework used in industry to perform predictive maintenance. Theses frameworks make ease to deploy a predictive maintenance solution. They do not necessary require a data science background.

Indeed, our case study is positioned in the frame of health monitoring systems. The final system should be able to increase the safety of the helicopter, the availability of the helicopter and reduces the maintenance costs. In contrast to the existing health monitoring systems, we propose to perform an *on-board* real-time detection using a manycore processor. Our study has the ambition to answer questions related to the transformation of a ground application to an on-board application which gradually computes health indicators using a manycore processor. Our design should be able to respect real-time constraints given by the volume of vibration data. Chapter 2, page 20 will detail the existing Health Monitoring System (HMS) and insists on the full chain: from the on-board data acquisition to the on-ground computation.

Finally, we need a computing platform that enables high performance computing capabilities because HMS involves signal processing algorithms which are known to be time consuming. We choose the manycore MPPA-256 to deploy our solution because it has properties which satisfy avionic constraints (high performance, energy efficient and predictability mechanisms). The overall architecture of the processor MPPA-256 is discussed in chapter 7, page 97. We have also compared our software architecture, timing measurements on the processor to the state-of-the art.

Chapter 11

Conclusion and Perspectives

11.1 Context of the Thesis

Modern avionic systems are using the Integrated Modular Avionics (IMA) standard where many applications with mixed criticality are hosted on the same platform. Moreover, there is a departure from single core processor to mutli/many-core processor in order to integrate more advanced functions.

Many-core processors are interesting candidate for avionic applications. Many-core processors have emerged as an evolution of multi-core processors. They offer a high computing capabilities, less Size, Weight and Power consumption (SWaP). Most of them are designed to achieve high performance computing, but offer no timing guarantees. However, the embedded systems market is not only interested in high performance but predictability mechanisms is a also crucial factor.

First, we select the many-core processor MPPA-256 from Kalray. Our choice is driven by its predictability mechanisms since the design, the high-performance computing capabilities and the low power consumption.

Second, the current Health Monitoring System (HMS) monitors the vibration of the rotating parts of the helicopter. It computes health indicators based on signal processing algorithms. The data analysis is performed when the helicopter is on-ground. However, this function reduces the availability of the helicopter which is not in service during ground analysis. In addition, the mechanical failures cannot be detected during the flight.

In fact, the HMS function is a good candidate to experiment on-board real-time computations using a many-core processor. The computation involves signal processing algorithms which require an important bandwidth. The HMS function should also raise alerts to the operators crew which require a bounded response time. Finally, the HMS function is appropriate to explore parallelism because it has different sensors which are functionally independent.

11.2 Summary

We have first analyzed the existing HMS function from the acquisition of the raw vibration signal to the extraction of health indicators. The computation of health indicators has four major steps: detecting the tops of the reference shaft revolutions, estimating the virtual tops of the monitored shaft revolutions, the interpolation step and the average signal. Then, since the current HMS is implemented in Java, we have designed our C reference version by coding the same textual specifications. Afterward, we have validated our C reference version by comparing different health indicators like OM1, RMSR with the Java ones. The results given by the two different methods are the same even if there implementations

are different for instance the C version does not require any external library for the linear interpolation function.

Then, we have transformed the on-ground C reference version, in which the global average is used to estimate the virtual tops of the monitored shaft, into an incremental embedded real-time computation of the same indicators. This transformation is performed using two different strategies: a growing or sliding window. Moreover, we have illustrated that the indicators calculated using a growing window are closer than the reference ones.

Since, the MPPA-256 has 2MB per internal computing cluster, we have estimated the required amount of memory to compute health indicators from raw vibration data to functional data in the computing cluster. The estimation is very fine-grain and takes into account the size of packet sent from acquisition unit to the MPPA-256, the sampling frequency, the speedup ratio, the interpolation size and the acquisition range of the speed of the rotor.

After this, we have investigated a potential bottleneck on the IO cluster. At this step, there is no computation in the internal cluster. We focus only on the IO cluster. The investigation is performed with two experiments: PCIe bandwidth and MPPA-256 latency experiments. The PCIe GEN3 provides a sufficient bandwidth to transmit the data from the Host to the IO Cluster. But, the main limitation is the data transfer between the IO and computing clusters. The latency experiment has defined a trade-off between the volume of input data and the latency. We defined VIBRATION_CHUNK_SIZE as the minimal size of packet to transmit from the acquisition unit to the MPPA-256. VIBRATION_CHUNK_SIZE is equal to 32KB.

Furthermore, we have designed a pipeline architecture to maximize the throughput. The proposed software architecture separates the sending and receiving functions in the Host, IO and computing clusters. There are also two threads in the internal clusters dedicated to the input/output flow. The different stages of our pipeline architecture exchange data using intermediate buffers. Since, the workers are both consumer and producer, a *producer/consumer* synchronization technique is implemented to ensure the data coherency between the different stages of the pipeline.

We ended our preliminary experiments with a workload estimation. Given a sampling frequency fs =31 kHz and a number of sensors of 256, one MPPA-256 cluster is able to address a workload of 8 FFT of 1024 samples.

Finally, we have implemented real-time health indicators on the many-core processor MPPA-256. The real-time computation of vibration data combined with a speed of the rotor which is cyclically changing make more challenging the respect of the WCET and the buffers dimensioning. Given an acquisition session, our mapping allows to respect the timing constraint. Our measurements consider the acquisition session that have the higher timing constraint: a speedup ratio of 75 and a sampling frequency fs = 31 kHz. In addition, we have shown that the timing constraints are still satisfied even if our mapping does not offer a perfect workload balancing. The causes of the burst processing are also mentioned: a non integer speedup ratio and the variation of the speed of the rotor.

11.3 Contributions

This thesis presents five main contributions:

- **Building an incremental version of the HMS function**: the first contribution covers the transformation of an on-ground reference computation of a health indicators, in which the global average *monitored_revolution_mean* is used to estimate the positions of the tops of the monitored shafts, into an incremental embedded real-time computation of the same indicators, in which the average is calculated using either a *growing* or a *sliding* window.

- **Management of the input flow**: the second contribution focuses on the management of the input flow provided by the sensors. We have illustrated how the input vibration data is read by chunk of size VIBRATION_CHUNK_SIZE thanks to the latency experiment, how the input flow is grouped per reference shaft revolutions thanks to the top detections. We have also addressed the earliest moment when the grouped data are available inside the internal cluster memory and when they are consumed by workers in order to process health indicators. In addition, the workload estimation of a given packet of size VIBRATION_CHUNK_SIZE transmitted from the acquisition unit to the MPPA-256 is also mentioned. At each step of the processing, the maximum amount of data to compute is calculated.

- **Building a Time-stamping tool**: the third contribution tackles a time-stamping tool to perform timing measurements on the target. We have designed our lightweight tracing mechanism, by adding timestamps to the flows of data. The additional data has not changed the functional behavior of our application. The time-stamping tool allows to measure execution duration, latency, bandwidth experiments, verify the predictability by addressing the variability of a given measurement repeated many times.

- **Building a software architecture which respects the real-time constraints**: In this contribution, we have proposed a software architecture that respect the timing constraints. Our measurements consider the maximum frequency and speedup ratio. The mapping covers the Host, IO and computing clusters. It is based on a pipeline architecture with three stages in the computing cluster.

- **Illustrating the limits of the HMS function**: the fifth contribution addresses the limits of the current function. We have illustrated the limits of the convergence criterion. Our experiments have shown that the contextual parameters of the helicopter like the NR speed must be correlated to the health indicators in order to reduce false alerts.

11.4 Future Work

11.4.1 Implement the whole HMS function

The on-board real-time HMS implemented in the scope of this thesis supports the processing of health indicators of a given acquisition session. It means, our implementation operates like the existing one: *one* sampling frequency per acquisition session which has different monitored shafts and speedup ratios with respect to the reference shaft.

However, *bearing* indicators are not yet implemented. Bearing rings and balls are not perfectly smooth as well as the inner and outer races. Vibrations produced by rolling bearing can result from imperfections during the manufacturing process (poor lubrication) , ambulance, misalignment ([95]). The bearing vibrations signatures can be analyzed using first a time waveform. The time waveform shows the impact rate or frequency. This frequency is called the *bearing tone*. The same information can be displayed using a spectrum analysis (one peak every time the ball is hitting the impact).

11.4.2 Towards using new sensor technologies

Since, the many-core MPPA-256 processor offers high performance computing capabilities, we have illustrated that it opens new horizons to explore *high frequency* sensors technologies in instead of piezoelectric sensors. The maximum sampling frequency in the existing HMS is $31kHz$. However, the ultrasonic sensors can go to 20 MHz and are widely used in the industry to monitor the vibration level of the

rotating components. The ultrasonic sensors allow to detect invisible alterations inside the material of the shaft.

11.4.3 Towards performing continuous on-board acquisitions

Acquisitions are not performed at anytime during the flight but at specific flight regime. Flight regimes are determined by using 2 contextual parameters of the helicopter: *IAS* (Indicator Air Speed) and ZRS (Radio Altitude). The current acquisitions are performed during the *SFL* (Steady Flight Level flight) flight regime. The SFL regime is a regime in which the helicopter is flying above a given altitude of ZRS (*ZRSmax*) and when the air speed is between *IASmin* and *IASmax*.

However, intermittent acquisitions cannot be compared because the contextual parameters are continuously changing. In order to perform acquisitions continuously at any flight regime, the contextual parameters and vibration data must be sampled at the same time. The idea is to compare indicators having the sames contextual parameters.

In addition, The reference HMS has various shafts which are sampled at *different* frequencies. Further work will be devoted to the implementation of the computations for several sensors, providing values at different sampling frequencies, and having different ratios with respect to the reference shaft.

11.4.4 Certification approach of multi/many-core processor in avionic industry

Certification is a set of activities that consists of proving via detailed documents that the developed product meets the requirements of a standard. These documents must indicate the assumptions, the input data, the results obtained, etc. The certification agreement is given by an independent organization.

The mean of compliance has to be reviewed with the recent and next generation of processors (multi/many-core processors). Then, the Certification Authorities Software Team (CAST) has provided recommendations and a set of guidance in the document CAST-32A position paper for software planning, development and verification in multi-core processor ([96]). It is motivated because the standards DO-178B/ED-12B and DO-178C /ED-12C are written to address the certification of software hosted on a single processor.

Due to resources sharing, interferences have first to be identified. Second, the applicant has to justify that those interferences are properly mitigated. The features provided by the MPPA-256 allow to drastically reduce the number of interferences and therefore reduce the certification effort ([97]). Each interference is a key certification issue and has to be identified and showing that it has no impact on the behavior of the application.

List of Publications

1. Moustapha Lo, Nicolas Valot, Florence Maraninchi and Pascal Raymond. Implementing a real-time avionic application on a many-core processor, 42nd European Rotorcraft Forum (ERF),2016.
2. Moustapha Lo, Nicolas Valot, Florence Maraninchi and Pascal Raymond. Real-time on-board manycore implementation of a health monitoring system: Lessons learnt, 9th European Congress Embedded Real-time Software and Systems, 2018.

Appendix A

Listings

Listing A.1: Gathering vibration samples of a given accelerometer in a array

```
1  double VibrationSample[NB_SAMPLE]; // vibration samples of a given accelerometer in the
       acquisition file
2  static void ComputeReferenceTop (UINT8_t id,ANY_t value){
3     // id: the pattern we search in the file
4     // value: the value of the pattern
5     static UINT32_t cpt_accel_in_range =0;
6     static UINT32_t cpt_accel_sensitivity  =0;
7     static UINT32_t cpt =0;
8     static UINT16_t NUMBER_ACCEL;
9     switch (id){
10    case PHASE: // read sensor phase values
11       //... Do some computation
12       break;
13    case NUMBER_ACTIVATED_ACCEL: // get the number of accelerometers in the
          acquisition file
14       NUMBER_ACCEL = ((UINT16_t)value);
15       break;
16    case ACCEL_SENSITIVITY: // Get accelerometer sensitivity
17       if( cpt_accel_sensitivity  ==SENSOR_ID){ // select one accelerometer from among all
             activated
18          accel_sensitivity  =(UINT16_t)value;
19       }
20        cpt_accel_sensitivity  ++;
21       break;
22    case ACCEL_IN_RANGE : // Get accelerometer range
23       if(cpt_accel_in_range==SENSOR_ID){ // select one accelerometer from among all
             activated
24          input_range=(UINT16_t)value;
25       }
26       cpt_accel_in_range ++;
27       break;
28    case SAMPLE_VALUE:
29       if(cpt%NUMBER_ACCEL==SENSOR_ID){ // Select an accelerometer and convert
             vibration sample in g unit
30          vibration_in_g =( input_range *((double)(SINT16_t)value)*2)/(((1<<16)-1)*(double)accel_sensitivity);
31          VibrationSample[nbVibrationSample]= vibration_in_g ;     // Gather vibration samples in a
                array
32          nbVibrationSample++;
33       }
34       cpt++;
35       break;
36    }
37 }
```

Listing A.2: Prototype of the method *interpolate* using the Java class *LinearInterpolator*

```
1   /**
2    * Computes a linear interpolating function for the data set.
3    *
4    * @param x
5    *        the arguments for the interpolation points
6    * @param y
7    *        the values for the interpolation points
8    * @return a function which interpolates the data set
9    */
10
11  public PolynomialSplineFunction interpolate (double x[], double y[]);
```

Listing A.3: Computing a linear interpolation of a given array of samples of a monitored shaft revolution

```
1  void LinearInterpolation (double *in, double in_size,double *out,int out_size)
2  {
3      /*
4
5      INPUTS:
6      in: raw array of samples of a monitored shaft revolution
7      in_size: size of the input array "in"
8
9      OUTPUTS:
10     out: array of interpolated samples
11     out_size: size of the output array "out" given by the spectification document
12     */
13
14     double mu, y1,y2, x,dy;
15     int i,j;
16     for (i=0;i<out_size;i++){
17         x = ( in_size / out_size )*i;  // interpolation step
18         j = (int)x;                    // point to interpolate
19         mu = x - ((int)x);             // calculate coefficients for the linear interpolation
               function
20         y1= in[j];
21         y2= in[j+1];
22         dy=y2-y1;
23         out[i]= y1+dy*mu;              // calculate linear interpolation
24     }
25 }
```

Listing A.4: Calculate FFT Complex inputs using real vibration data

```
#define REAL 0
#define IMAG 1

void generate_input_fft (double *in, fftw_complex* out, int size) {
/*
INPUTS:
  -in: real samples that represent the average signal
  -size: size of the array "in"

OUTPUT:
  -out: array of complex fft inputs
     */

    int i;
    for (i = 0; i < size ; i++) {
        out[i][REAL]=in[i];
        out[i][IMAG] = 0;
    }
}
```

Listing A.5: Computing the FFT module

```
1  void Computemodule(fftw_complex* result, int length,double *module) {
2
3      int i;
4      for (i = 0; i < length; i++) {
5          module[i] = sqrt ( result [i][REAL]*result[i][REAL] + result[i][IMAG]*result[i][IMAG]);
6      }
7  }
```

Listing A.6: Computing a Fast Fourier Transform using FFTW library

```
1  #define REAL 0
2  #define IMAG 1
3
4  void ComputeFFT(double *real, fftw_complex *complex_fft, int fft_length){
5      fftw_complex  inputfft [ fft_length ];
6      int i;
7
8      // Initialization, Create a plan
9      fftw_plan  plan = fftw_plan_dft_1d( fft_length ,
10                 inputfft ,
11                 complex_fft,
12                 FFTW_FORWARD,
13                 FFTW_ESTIMATE);
14
15     // Create FFT inputs
16      generate_input_fft ( real , inputfft , fft_length );
17
18
19     // Compute FFT using FFTW library
20     fftw_execute(plan);
21
22        // Normalize FFT results
23     for(i=0; i< fft_length ; i++) {
24         complex_fft[i][REAL] = complex_fft[i][REAL]/fft_length;
25         complex_fft[i][IMAG] = complex_fft[i][IMAG]/fft_length;
26     }
27
28     fftw_destroy_plan(plan);
29 }
```

Listing A.7: Computing the RMS of an array of samples with arbitrary size

```
1  /*
2  -INPUTS:
3          - array of samples "data"
4          - size: IT samples (size of the array "data")
5
6  -OUTPUT:
7          - rms value: root mean square
8
9  */
10
11 void RMS(double *data, int size, double *rms){
12     double mean; // mean
13     double variance; // variance
14
15     // Compute Mean of the array "data"
16     computeMean(data,size, &mean);
17
18     // Compute Variance of the array "data"
19     computeVariance(data, size,&mean,&variance);
20
21     //Compute RMS(Root mean square) of "data"
22     *rms=sqrt( variance );
23 }
```

Listing A.8: Computing the RMS Residual indicator

```
1  /*
2  INPUTS:
3    -fft: complex output results of a FFT
4    -fft_length: size of the array "fft" in complex
5    -remove: array containing the index of harmonics to be removed from the spectrum
         and their multiples indices
6    -nb_harmonics_to_remove: size of the array "remove"
7
8  OUTPUT:
9    -rms: rms residual
10
11 */
12
13 void computeRMSR(fftw_complex *fft,int fft_length,int *remove,int nb_harmonics_to_remove,double *rms){
14
15      fftw_complex  residual [IT];
16      fftw_complex  reversefftCplx [IT];
17      double reversefftReal[IT];
18
19      // Compute complex FFT residual
20      residualfftComplex ( fft , fft_length ,remove,nb_harmonics_to_remove,residual);
21
22      //Compute reverse FFT
23      ComputeReverseFFT(residual, fft_length , reversefftCplx );
24
25      //Transform reversefftCplx to reversefftReal
26      int i;
27      for(i=0; i< fft_length ; i++)  reversefftReal [i]= reversefftCplx [i ][REAL];
28
29      // Compute RMS on the residual spectrum
30      RMS(reversefftReal, fft_length ,rms);
31
32 }
```

Listing A.9: Computing the residual spectrum

```
1  void residualfftComplex(fftw_complex * fft , int fft_length ,int *lines, int length_list ,fftw_complex * residual ){
2
3      /* INPUTS:
4       * -fft: complex fft already computed
5       * -fft_length: size of "fft" array in complex
6       * -lines: list of harmonics to be removed from the spectrum and their multiple
            indices
7       * -length_list: size of the array "lines"
8       *
9       *
10      * OUTPUT:
11      * -residual: complex residual spectrum
12      * */
13
14     int index_fft;
15     int j;
16     int nb_harmonics=0;
17
18     // residual initialization
19
20     for(index_fft=0; index_fft < fft_length ; index_fft ++){
21          residual [ index_fft ][REAL]=fft[index_fft ][REAL] ;
22          residual [ index_fft ][IMAG]=fft[index_fft ][IMAG];
23
24     }
25
26     // Remove a list of fft harmonics
27     while (nb_harmonics<length_list){
28
29         int index_harmonic = lines[nb_harmonics];
30         for (j = 1; j <= ( fft_length / (2 * index_harmonic)); ++j) {
31             if ((j * index_harmonic + 1) != fft_length / 2) {
32                     // set to 0 the harmonic at index j
33                 residual [j*index_harmonic][REAL] = 0;
34                 residual [j*index_harmonic][IMAG] = 0;
35                     // set to 0 the conjugate harmonic
36                 residual [ fft_length - j*index_harmonic][REAL] = 0;
37                 residual [ fft_length - j*index_harmonic][IMAG] = 0;
38             }
39         }
40         nb_harmonics++;
41     }
42 }
```

Listing A.10: Computing the reverse FFT of the residual spectrum

```
1  void ComputeReverseFFT(fftw_complex *residual,int fft_length, fftw_complex * reversefft ){
2
3      // Initialization, create a plan to calculate a reverse FFT
4      fftw_plan plan = fftw_plan_dft_1d( fft_length ,
5                  residual ,
6                  reversefft ,
7                 FFTW_BACKWARD,
8                 FFTW_ESTIMATE);
9
10     // Compute reverse FFT
11     fftw_execute(plan);
12
13        // destroy plan
14     fftw_destroy_plan(plan);
15 }
```

Listing A.11: Calculating an incremental variance

```
1  void IncrementalVariance(double input, double previous_M1, double current_M1, int nb_samples_rev, double *M2){
2
3  /* INPUTS : new sample
4  * *previous_M1: previous mean
5  * *previous_S : N*variance
6  * *current_M1: current mean
7  * *current_S
8  * *M2: variance
9  * nb_samples_rev: number of vibration samples of the current monitored shaft revolution
10 * */
11
12
13     double current_S=0;
14     static double previous_S=0;
15     static int nb_samples_start=0; // counter of samples since the beginning of the acquisition
           session
16
17     nb_samples_start++;              // new sample
18
19     current_S= previous_S+(input-previous_M1)*(input-current_M1);
20
21     *M2= current_S / nb_samples_start ;  // incremental variance formula
22
23     previous_S=current_S;
24 }
```

Bibliography

[1] Pierre Bieber, Frédéric Boniol, Marc Boyer, Eric Noulard, and Claire Pagetti. New challenges for future avionic architectures. *AerospaceLab*, (4):p–1, 2012.

[2] Wtilesmachinery. Corrective versus preventive maintenance: What is the difference and where is the value? https://www.stilesmachinery.com/articles/corrective-versus-preventive-maintenance-what-is-the-difference-and-where-is-the-value, 2018.

[3] Christopher B Watkins and Randy Walter. Transitioning from federated avionics architectures to integrated modular avionics. In *Digital Avionics Systems Conference, 2007. DASC'07. IEEE/AIAA 26th*, pages 2–A. IEEE, 2007.

[4] James W.Ramsey. Integrated modular avionics: Less is more. http://www.aviationtoday.com/2007/02/01/integrated-modular-avionics-less-is-more/, 2 2007.

[5] Yannick Moy, Emmanuel Ledinot, Hervé Delseny, Virginie Wiels, and Benjamin Monate. Testing or formal verification: Do-178c alternatives and industrial experience. *IEEE software*, 30(3):50–57, 2013.

[6] Jan Nowotsch, Michael Paulitsch, Daniel Bühler, Henrik Theiling, Simon Wegener, and Michael Schmidt. Multi-core interference-sensitive wcet analysis leveraging runtime resource capacity enforcement. In *Real-Time Systems (ECRTS), 2014 26th Euromicro Conference on*, pages 109–118. IEEE, 2014.

[7] Hamza Rihani. *Many-Core Timing Analysis of Real-Time Systems*. PhD thesis, Université Grenoble Alpes, 2017.

[8] WikiHow. Comment déterminer un rapport de transmission. https://fr.wikihow.com/d%C3%A9terminer-un-rapport-de-transmission, 2018.

[9] Wikipedia. Accelerometer. https://en.wikipedia.org/wiki/Accelerometer#Types_of_accelerometer, 2 2018.

[10] Bodycote. Main gear box. http://www.bodycote.com/en/markets/aerospace-and-defense/helicopter/main-gear-box.aspx, 2018.

[11] brainkart. Gear box: Principle of gearing and types of gear boxes. http://www.brainkart.com/article/Gear-Box--Principle-of-Gearing-and-Types-of-Gear-Boxes_5054/, 2018.

[12] Yong Shen, Tianmin Shan, Xingwang Li, Huiyun Wang, Canfei Sun, and Yifei He. A method of health evaluation for the gear of helicopter main gearbox. In *Prognostics and System Health Management Conference (PHM-Chengdu), 2016*, pages 1–5. IEEE, 2016.

[13] William H Press, Saul A Teukolsky, William T Vetterling, and Brian P Flannery. *Numerical recipes in C*, volume 2. Cambridge university press Cambridge, 1996.

[14] Sereema Lab. Rotor imbalance detection from automated 1p analysis and measurement : Real case study during a long period for different large size wtg. 2014.

[15] Lufti Arebi, Fengshou Gu, and Andrew Ball. Rotor misalignment detection using a wireless sensor and a shaft encoder. University of Huddersfield, 2010.

[16] A Majidian and MH Saidi. Comparison of fuzzy logic and neural network in life prediction of boiler tubes. *International Journal of Fatigue*, 29(3):489–498, 2007.

[17] Raquel Lazcano, Daniel Madroñal, Rubén Salvador, Karol Desnos, Maxime Pelcat, Raùl Guerra, Himar Fabelo, Samuel Ortega, Sebastián López, Gustavo Marrero Callicó, et al. Porting a pca-based hyperspectral image dimensionality reduction algorithm for brain cancer detection on a manycore architecture. *Journal of Systems Architecture*, 77:101–111, 2017.

[18] Wolfram MathWorld. Floor function. `http://mathworld.wolfram.com/FloorFunction.html`, 02 2018.

[19] Wikipedia. Window function. `https://en.wikipedia.org/wiki/Window_function`, 8 2017.

[20] Saad Ahmad. Fft spectral leakage and windowing. `http://saadahmad.ca/fft-spectral-leakage-and-windowing/`, March 2015.

[21] Michael V Cook. *Flight dynamics principles: a linear systems approach to aircraft stability and control*. Butterworth-Heinemann, 2012.

[22] helmets. What is a "g". `https://helmets.org/g.htm`, October 2017.

[23] Fresh2Refresh. C – round() function. `https://fresh2refresh.com/c-programming/c-arithmetic-functions/c-round-function/`, 2018.

[24] Apache. Class linearinterpolator. `http://commons.apache.org/proper/commons-math/javadocs/api-3.6/org/apache/commons/math3/analysis/interpolation/LinearInterpolator.html`, 2016.

[25] Paul Bourke. Interpolation methods. `http://paulbourke.net/miscellaneous/interpolation/`, December 1999.

[26] FFTW. Complex one-dimensional transforms tutorial. `http://www.fftw.org/fftw2_doc/fftw_2.html`, 2018.

[27] Springer. Discrete fourier transform. `http://www.springer.com/cda/content/document/cda_downloaddocument/9781402066283-c2.pdf?SGWID=0-0-45-1119940-p173753625`, 2010.

[28] Slobodan Stupar, Aleksandar Simonovic, and Miroslav Jovanovic. Measurement and analysis of vibrations on the helicopter structure in order to detect defects of operating elements. *Scientific Technical Review*, 62(1):58–63, 2012.

[29] Donald E Knuth, Hiroaki Saitou, Takahiro Nagao, Shougo Matui, Takao Matui, and Hitoshi Yamauchi. *of Book: The Art of Computer Programming.-Volume 2, Seminumerical Algorithms (Japanese Edition)*, volume 2. ASCII, 2004.

[30] Tony Finch. Incremental calculation of weighted mean and variance. *University of Cambridge*, 4:11–5, 2009.

[31] Wikipedia. Algorithms for calculating variance. `https://en.wikipedia.org/wiki/Algorithms_for_calculating_variance`, March 2018.

[32] Robert R Schaller. Moore's law: past, present and future. *IEEE spectrum*, 34(6):52–59, 1997.

[33] Wikipedia. Moore's law. `https://en.wikipedia.org/wiki/Moore%27s_law`, 7 2015.

[34] Fred J Pollack. New microarchitecture challenges in the coming generations of cmos process technologies (keynote address). In *Proceedings of the 32nd annual ACM/IEEE international symposium on Microarchitecture*, page 2. IEEE Computer Society, 1999.

[35] Antonio Paolillo, Paul Rodriguez, Nikita Veshchikov, Joël Goossens, and Ben Rodriguez. Quantifying energy consumption for practical fork-join parallelism on an embedded real-time operating system. In *Proceedings of the 24th International Conference on Real-Time Networks and Systems*, pages 329–338. ACM, 2016.

[36] Dawei Li, Jie Wu, Keqin Li, and Kai Hwang. Energy-aware scheduling on multiprocessor platforms with devices. In *Cloud and Green Computing (CGC), 2013 Third International Conference on*, pages 26–33. IEEE, 2013.

[37] Nikolai Sakharnykh. Tridiagonal solvers on the gpu and applications to fluid simulation. In *NVIDIA GPU Technology Conference*, 2009.

[38] Meteo France. Simuler l'évolution du climat. `http://www.meteofrance.fr/videos/climat/49389-simuler-l-evolution-du-climat`, 2009.

[39] top500. Top 500 supercomputers. `top500.org`, november 2017.

[40] András Vajda. *Programming many-core chips*. Springer Science & Business Media, 2011.

[41] Selma Saidi, Rolf Ernst, Sascha Uhrig, Henrik Theiling, and Benoît Dupont de Dinechin. The shift to multicores in real-time and safety-critical systems. In *Proceedings of the 10th International Conference on Hardware/Software Codesign and System Synthesis*, pages 220–229. IEEE Press, 2015.

[42] Reetuparna Das, Rachata Ausavarungnirun, Onur Mutlu, Akhilesh Kumar, and Mani Azimi. Application-to-core mapping policies to reduce memory interference in multi-core systems. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, pages 455–456. ACM, 2012.

[43] Hamza Rihani, Matthieu Moy, Claire Maiza, Robert I Davis, and Sebastian Altmeyer. Response time analysis of synchronous data flow programs on a many-core processor. In *Proceedings of the 24th International Conference on Real-Time Networks and Systems*, pages 67–76. ACM, 2016.

[44] Mateusz Berezecki, Eitan Frachtenberg, Mike Paleczny, and Kenneth Steele. Many-core key-value store. In *Green Computing Conference and Workshops (IGCC), 2011 International*, pages 1–8. IEEE, 2011.

[45] Bo Yuan, Jin Dou Fan, and Bin Liu. Cooperative mechanism of local memory and cache in network processors. In *Applied Mechanics and Materials*, volume 380, pages 1969–1972. Trans Tech Publ, 2013.

[46] Manel Fakhfakh. *Réconcilier performance et prédictibilité sur un many-coeur en utilisant des techniques d'ordonnancement hors-ligne*. PhD thesis, Paris 6, 2014.

[47] Andreas Olofsson. Epiphany-v: A 1024 processor 64-bit risc system-on-chip. *arXiv preprint arXiv:1610.01832*, 2016.

[48] Andreas Olofsson, Tomas Nordström, and Zain Ul-Abdin. Kickstarting high-performance energy-efficient manycore architectures with epiphany. *arXiv preprint arXiv:1412.5538*, 2014.

[49] Jim Held. Single-chip cloud computer. In *an IA Tera-Scale Research Processor. In: Guarracino, MR, Vivien, F., Träff, JL, Cannatoro, M., Danelutto, M., Hast, A., Perla, F., Knüpfer, A., Di Martino, B., Alexander, M.(eds.) Euro-Par-Workshop*, page 85, 2010.

[50] Jason Howard, Saurabh Dighe, Yatin Hoskote, Sriram Vangal, David Finan, Gregory Ruhl, David Jenkins, Howard Wilson, Nitin Borkar, Gerhard Schrom, et al. A 48-core ia-32 message-passing processor with dvfs in 45nm cmos. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International*, pages 108–109. IEEE, 2010.

[51] Leonardo Dagum and Ramesh Menon. Openmp: an industry standard api for shared-memory programming. *IEEE computational science and engineering*, 5(1):46–55, 1998.

[52] William Gropp, Rajeev Thakur, and Ewing Lusk. *Using MPI-2: Advanced features of the message passing interface*. MIT press, 1999.

[53] B.D. de Dinechin, R. Ayrignac, P.-E. Beaucamps, P. Couvert, B. Ganne, P.G. de Massas, F. Jacquet, S. Jones, N.M. Chaisemartin, F. Riss, and T. Strudel. A clustered manycore processor architecture for embedded and accelerated applications. In *High Performance Extreme Computing Conference (HPEC), 2013 IEEE*, pages 1–6, Sept 2013.

[54] Benoît Dupont De Dinechin, Duco Van Amstel, Marc Poulhiès, and Guillaume Lager. Time-critical computing on a single-chip massively parallel processor. In *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014*, pages 1–6. IEEE, 2014.

[55] Theo Ungerer, Christian Bradatsch, Mike Gerdes, Florian Kluge, Ralf Jahr, Jörg Mische, Joao Fernandes, Pavel G Zaykov, Zlatko Petrov, Bert Böddeker, et al. parmerasa–multi-core execution of parallelised hard real-time applications supporting analysability. In *Digital System Design (DSD), 2013 Euromicro Conference on*, pages 363–370. IEEE, 2013.

[56] Richard L Alena, John P Ossenfort, Kenneth I Laws, Andre Goforth, and Fernando Figueroa. Communications for integrated modular avionics. In *Aerospace Conference, 2007 IEEE*, pages 1–18. IEEE, 2007.

[57] Paul J Prisaznuk. Arinc 653 role in integrated modular avionics (ima). In *2008 IEEE/AIAA 27th Digital Avionics Systems Conference*, pages 1–E. IEEE, 2008.

[58] Aeronautical Radio Inc. Arinc 664: Aircraft data network, part 1: Systems concepts and overview. 2002.

[59] Solen Cochard. Étude des bus avioniques. *Travail d'Étude et de Recherche UBO*, 2002.

[60] Aamir Mairaj and Rohail Tahir. Swap reduction: vital for choice of avionics architecture. Pakistan Aeronautical Complex, Avionics Production Factory. Kamra, Pakistan: International Conference on Engineering and Emerging Technologies, 2014.

[61] Quentin Perret, Pascal Maurère, Éric Noulard, Claire Pagetti, Pascal Sainrat, and Benoit Triquet. Temporal isolation of hard real-time applications on many-core processors. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–11, April 2016.

[62] Gary Graunke and Shreekant Thakkar. Synchronization algorithms for shared-memory multiprocessors. In *Programming languages for parallel processing*, pages 26–35. IEEE Computer Society Press, 1995.

[63] Peter Marwedel. *Embedded system design*, volume 1. Springer, 2006.

[64] David R Butenhof. *Programming with POSIX threads*. Addison-Wesley Professional, 1997.

[65] Mathieu Desnoyers and Michel Dagenais. Lttng: Tracing across execution layers, from the hypervisor to user-space. In *Linux symposium*, volume 101, 2008.

[66] Thomas Carle, Manel Djemal, Dumitru Potop-Butucaru, Robert De Simone, and Zhen Zhang. Static mapping of real-time applications onto massively parallel processor arrays. In *Application of Concurrency to System Design (ACSD), 2014 14th International Conference on*, pages 112–121. IEEE, 2014.

[67] Ravi Budruk, Don Anderson, and Tom Shanley. *PCI express system architecture*. Addison-Wesley Professional, 2004.

[68] tested.com. Theoretical vs. actual bandwidth: Pci express and thunderbolt. `http://www.tested.com/tech/457440-theoretical-vs-actual-bandwidth-pci-express-and-thunderbolt/`, 9 2013.

[69] Alex Goldhammer and John Ayer Jr. Understanding performance of pci express systems. *Xilinx WP350, Sept*, 4, 2008.

[70] M Vaidehi and TR Nair. Multicore applications in real time systems. *arXiv preprint arXiv:1001.3539*, 2010.

[71] Trishul M Chilimbi, Bob Davidson, and James R Larus. Cache-conscious structure definition. In *ACM SIGPLAN Notices*, volume 34, pages 13–24. ACM, 1999.

[72] Soeren Suesse. Dynamic rotor blade displacement tracking with fiber-optical sensors for a health and usage monitoring system. In *18th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference*, page 3659, 2017.

[73] Wikipedia. Facteur de charge (aérodynamique). `https://fr.wikipedia.org/wiki/Facteur_de_charge_(a%C3%A9rodynamique)`, Mars 2018.

[74] Avionics. Hums technology. `https://www.aviationtoday.com/2012/05/01/hums-technology/`, May 2012.

[75] Sikorsky. Sikorsky, phi, metro demo real-time hums on s-92. https://www.aviationtoday.com/2017/03/07/sikorsky-phi-metro-demo-real-time-hums-s-92/, 2017.

[76] ainonline. Real-time hums goes operational on phi's s-92. https://www.ainonline.com/aviation-news/business-aviation/2017-03-07/real-time-hums-goes-operational-phis-s-92, March 2017.

[77] Statistics how to. Regression analysis. http://www.statisticshowto.com/probability-and-statistics/regression-analysis/, 2018.

[78] Elizabeth A Cudney, Kioumars Paryani, and Kenneth M Ragsdell. Applying the mahalanobis–taguchi system to vehicle handling. *Concurrent Engineering*, 14(4):343–354, 2006.

[79] Wikipedia. Predictive maintenance. https://en.wikipedia.org/wiki/Predictive_maintenance, July 2011.

[80] Accelix Community Accelix Community Accelix community. What is predictive maintenance? https://www.accelix.com/community/predictive-maintenance/predictive-maintenance-explained/.

[81] Stephen Marsland. *Machine learning: an algorithmic perspective*. Chapman and Hall/CRC, 2011.

[82] Ronan Collobert, Koray Kavukcuoglu, and Clément Farabet. Torch7: A matlab-like environment for machine learning. In *BigLearn, NIPS workshop*, number EPFL-CONF-192376, 2011.

[83] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Mass storage systems and technologies (MSST), 2010 IEEE 26th symposium on*, pages 1–10. Ieee, 2010.

[84] Matei Zaharia, Reynold S Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J Franklin, et al. Apache spark: a unified engine for big data processing. *Communications of the ACM*, 59(11):56–65, 2016.

[85] Airports Council International North America. Industry white paper aircraft operating and delay cost per enplanement. Technical report, 2014.

[86] Amiral Technologies. À propos d'amiral technologies. https://www.amiraltechnologies.com/fr/a-propos/, 2018.

[87] Lab Horizions. Amiral technologies met l'intelligence artificielle au service de la maintenance industrielle. https://www.labhorizons.net/actualites/breves.php?id=7847, Août 2018.

[88] Daniel Madroñal, Raquel Lazcano, Rubén Salvador, Himar Fabelo, Samuel Ortega, Gustavo Marrero Callicó, E Juarez, and César Sanz. Svm-based real-time hyperspectral image classifier on a manycore architecture. *Journal of Systems Architecture*, 80:30–40, 2017.

[89] Johan AK Suykens and Joos Vandewalle. Least squares support vector machine classifiers. *Neural processing letters*, 9(3):293–300, 1999.

[90] S Kabwama, D Bulters, H Bulstrode, H Fabelo, S Ortega, GM Callico, B Stanciulescu, R Kiran, D Ravi, A Szolna, et al. Intra-operative hyperspectral imaging for brain tumour detection and delineation: Current progress on the helicoid project. *International Journal of Surgery*, 36:S140, 2016.

[91] Wei-Tsun Sun, Hugues Cassé, Christine Rochange, Hamza Rihani, and Claire Maıza. Using execution graphs to model a prefetch and write buffers and its application to the bostan mppa.

[92] Johan AK Suykens, Tony Van Gestel, Joos Vandewalle, and Bart De Moor. A support vector machine formulation to pca analysis and its kernel version. *IEEE Transactions on neural networks*, 14(2):447–450, 2003.

[93] Xilinx Virtex. Fpga family, 7.

[94] Takuya Kitamoto. Approximate eigenvalues, eigenvectors and inverse of a matrix with polynomial entries. *Japan journal of industrial and applied mathematics*, 11(1):73–85, 1994.

[95] SJ Lacey. An overview of bearing vibration analysis. *Maintenance & asset management*, 23(6):32–42, 2008.

[96] Irune Agirre, Jaume Abella, Mikel Azkarate-Askasua, and Francisco J Cazorla. On the tailoring of cast-32a certification guidance to real cots multicore architectures. In *Industrial Embedded Systems (SIES), 2017 12th IEEE International Symposium on*, pages 1–8. IEEE, 2017.

[97] Pierre Bieber, Frédéric Boniol, Youcef Bouchebaba, Julien Brunel, Claire Pagetti, Olivier Poitou, Thomas Polacsek, Luca Santinelli, and Nathanaël Sensfelder. A model-based certification approach for multi/many-core embedded systems. In *ERTS 2018*, 2018.