Real-time on-Board Manycore Implementation of a Health Monitoring System: Lessons Learnt

Moustapha Lo, Nicolas Valot
Airbus Helicopters
Florence Maraninchi, Pascal Raymond
Univ. Grenoble Alpes, CNRS, Grenoble INP*, VERIMAG, 38000 Grenoble, France

***Abstract*—Maintenance has long been a predominant activity in the industrial sector. Measuring and analyzing physical signals on the machines allows to provide a diagnosis on their health state. The more recent Health Monitoring Systems (HMS) allow to optimize the maintenance operations by performing *preventive* maintenance. The existing HMS are based on various signal-processing algorithms applied to vibration data gathered during flights, in order to compute health indicators. The computation of the indicators is done on-ground, once a full data set has been offloaded.**

In this paper, we report on experiments made to turn these *on-ground computations* into *on-board real-time computations*, using a many-core processor. There are two main issues to be addressed: (i) the management of the flow of inputs from sensors; (ii) the (hopefully tolerable) errors we make when transforming an on-ground algorithm that can treat data globally, into an on-board real-time algorithm that is necessarily incremental. We show that the error is indeed acceptable.

***Index terms*—** on-ground, on-board, real-time, many-core, global or incremental algorithms, Health Monitoring Systems.

I. Introduction

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 are 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 interpretation of the indicators in terms of mechanical defects, and the very choice of the indicators to be computed, are based on previous expertise and empirical studies of the helicopters.

This work was supported by the CAPACITES research project, supported by the French authorities through the "Investissements d'Avenir" program.

* Institute of Engineering Univ. Grenoble Alpes

The current implementation of the HMS is not embedded in the helicopter. 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 algorithms that define the various health indicators are computed on-ground. This requires a huge storage capacity, and a high network bandwidth for data offloading.

According to [7], the helicopter S-92 designed by Sikorsky is, since March, 2017, the first helicopter that has the ability to transmit in real-time raw vibration data. Vibration data are transmitted to a ground support team that performs maintenance operations. In others words, the real-time implementation is *not* embedded in the helicopter. It is a real-time but on-ground computation.

We study how to provide an on-line embedded version of the existing on-ground HMS application for several reasons. First, the existing application runs on the PC of the helicopter operator/customer, and cannot be trusted entirely by the helicopter manufacturer, who has to recommend periodic and potentially costly maintenance sessions. If the HMS is computed on-line, it will inherit the criticality level of the on-board computing unit, and be more reliable for the manufacturer. It is one of the enablers to replace periodic maintenance with predictive maintenance. Second, it will save time between flights, and therefore increase the helicopter availability.

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 and the ability to be used in certified equipment. According to [1], the benefits of the MPPA family of processors for critical real-time systems are: predictable computation and responses times, low power, and high performance.

The first challenge is the need to transform a global algorithm into an *incremental* one. With the on-ground global algorithms, we consider the full data set to compute indicators (e.g., we may use a global average). The embedded real-time 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 transformation of the algorithms will necessarily introduce some errors, w.r.t. the on-ground global algorithm considered as the reference implementation.

The second challenge is the real-time aspect: we focus on inputs, 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.

In this paper we are not concerned by the techniques that can be used to map the entire computations onto the computing units of the many-core processor. We carefully examine the whole chain between sensor data and the indicators, through the acquisition card, the bus connecting it to the processor, and the I/O interface of the processor. We perform experiments with a simple mapping, and derive various dimensioning indications, related to the number of sensors, the sampling frequency (i.e., the volume of data), and the number of indicators to be computed.

Section II describes the mechanical system, the sensors, and the maintenance decisions. Section III details the different steps of the existing global algorithm used on-ground to compute health indicators. Section IV explains the problems faced when transforming the existing algorithms into real-time embedded ones. Section V focuses on the architecture of the many-core processor MPPA-256. Section VI reports on our implementation experiments on the Kalray MPPA processor and finally section VII lists lessons learnt and further work.

II. Mechanical System, Acquisition Unit, and Indicators Analysis

A. Mechanical System

The mechanical system involves phase sensors and accelerometers. Phase sensors are placed 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 rest of the system. Various accelerometers are placed on the rotating elements to monitor their vibrations. Each *monitored shaft* has a fixed rotational speed ratio w.r.t. the speed of the reference shaft it depends on.

Figure 1 shows two reference shafts: the main and the tail rotors. Two monitored shafts are connected to the main rotor: IGB and OGB, with a rotational speed ratio $r_m = 2$ (they rotate twice faster). Two monitored shafts are connected to the tail rotor, TDS1 (Tail Drive Shaft left) and TDS2 (Tail Drive Shaft right), with a rotational speed ratio $r_t = 4$ (they rotate 4 times faster).

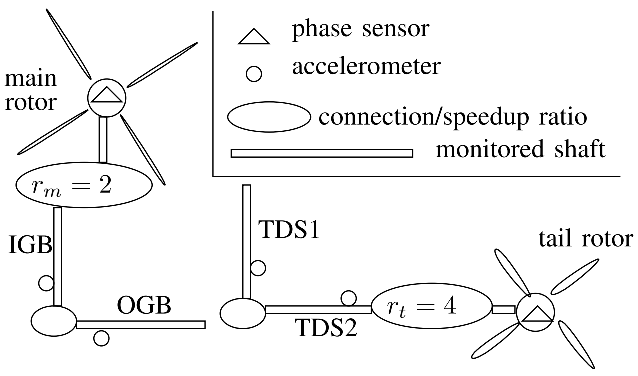

Fig. 1: The Mechanical System and the sensors

B. Acquisition Unit

The acquisition unit is a piece of equipment that acquires the sensor data, and provides a flow of inputs for the computations. During a flight, it performs one or several acquisition *sessions*. A session is characterized by the choice of enabled accelerometers, the sampling frequency, and the sampling size (number of samples per accelerometer acquired during the session). Each 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 for 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 are not delivered one sample at a time. The acquisition unit fills a buffer (the size of which corresponds to 100 ko, typically). The buffer is then read by the computation part, sufficiently often so as not to lose samples. In the on-line version, the size of the data packets exchanged between the acquisition card and the many-core processors has to be large enough, to ensure that the over-cost of the transmission w.r.t. to the payload is not too big. We examine this point in section VI-B.

C. Computation of the HMS Indicators and Maintenance Decisions

For a given acquisition file and monitored shaft, the existing off-line application computes a set of frequency indicators (OM1, OM2, ...) and temporal indicators (RMS,RMSR,Kurtosis, Skewness, ...). For instance, OM1 allows to detect the misalignment of the monitored shaft (i.e., the relative position deviation of the shaft from the collinear rotation axis). 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 illustrates the evolution of the indicator OM1 depending on the acquisition date. The x axis represents the acquisition dates from 1 to 5, and the y axis gives the indicator OM1 (in g unit).

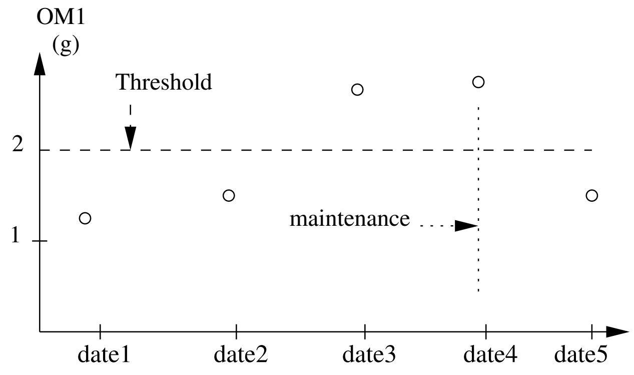

Fig. 2: Evolution of indicator OM1, for a given flight regime.

The human operators are provided with a web interface that essentially shows pictures similar to that of Fig. 2. A *threshold* value is set, based on human expertise. It may depend on a particular helicopter and operating conditions. If the indicator is above the threshold, it means a defect has been detected. In figure 2, 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 decided based on that observation, and performed after date 4. This is the reason why the OM1 indicator then decreases to $1.5g$ at date 5.

There is a trend towards using machine learning techniques in order to replace the human diagnosis based on indicators by automatic decision methods [5]. The machine learning algorithms should be trained with existing data from previous flights, and corresponding recorded human decisions.

D. Evaluation Criteria for the Online Version

When transforming the on-ground algorithms into embedded real-time ones, we need to compare the new version to a 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. The ultimate criteria is that, for the same data, the on-line version leads to the same decisions as the on-ground global version.

However, in the experiments reported in this paper, we cannot only look at the final result, because we cannot ask the human operators what they would have decided based on the results we compute. Instead, we carefully tracked all the steps of the algorithms (from the acquisition of sensor data to the computation of the indicators), in order to detect potential discrepancies as early as possible in the complete chain. This provides general guidelines on what to check when transforming any on-ground global algorithm into on-line real-time programs. The differences between the two versions my have different impact depending on the nature of the indicators (temporal of frequency domains, typically).

III. Existing global algorithm

The existing global algorithm uses as inputs the vibration data acquired during the flight. The acquisition file contains vibration samples of accelerometers and the tops of the phase sensor. The global algorithm is used to calculate the raw signal that will be used as inputs to compute the indicators.

A. Steps of the Global algorithm

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

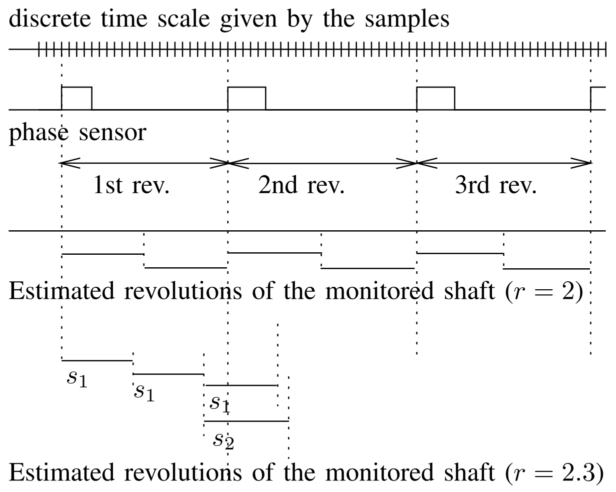
discrete time scale given by the samples
phase sensor
1st rev.
2nd rev.
3rd rev.
Estimated revolutions of the monitored shaft ($r = 2$)
s_1
s_1
s_1
s_2
Estimated revolutions of the monitored shaft ($r = 2.3$)

Fig. 3: Estimated tops of the monitored shaft with integer or non integer ratational speed ratios.

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. This allows 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.

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). It is important to notice that, when estimating the beginning and end of a monitored shaft revolution, we compute the position as a integer index on this time scale. This involves some rounding operations. Previous experiments have shown that the overall computation of the indicators is not too sensitive to these rounding operations, with the existing application. We will anyway reproduce the same computations of the indices in the on-line application, using the same rounding operations.

1) Detecting the tops of the reference shaft: The principle of a phase sensor is illustrated on the figure 3. 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, 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.

2) 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 1).

On figure 3, 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. This involves some rounding operations.

3) 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 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.

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.

In our objective of respecting the global reference algorithm, we must notice that this use of the global average is a source of discrepancy between the two versions. We will replace it by the average *until now*, which can be computed incrementally. The validation criteria mentioned in Section II-D is that this difference does not have too much impact on the final decision (through the complete chain: placement of the monitored virtual tops, linear interpolation, computation of the average signal for one revolution, and finally computation of the indicators and their presentation to the human decision).

4) Interpolation: In the previous steps, we have determined the beginning and end of each monitored shaft revolution. They do not have the same number of samples. A linear interpolation is performed, in order to get a fixed number of samples.

5) Synchronous average: The preparation of data ends with the computation of the average of all these interpolated revolutions, which gives the signal to be analyzed.

6) Indicators: Finally, the frequency and temporal indicators can be computed on this average signal.

B. Versions of the algorithm and summary of the potential sources of discrepancies

In our experiments, we consider several versions of the algorithms:

1) The Java implementation of the current on-ground HMS function, of which we only have the binary
2) A version in C, obtained by recoding the textual specification (still using the global average speed in order to determine the monitored shaft revolutions, as explained in section III-A3)
3) A version in C prepared for the embedded implementation, obtained by modifying the previous one as little as possible (but having to replace the global average by an average "*until now*")
4) The C code that will run on the embedded platform, which must take into account the constraints due to the connection of the embedded computing platform to the acquisition card.

The main potential source of discrepancies between those 4 versions is the use of the global or incremental average speed to estimate the position of the tops of the monitored shaft. We will focus on this source of errors in the selection of tests to be performed. Typically, this error will be bigger for acquisition sessions in which the speeds varies a lot.

Concerning the potential rounding errors for the placement of the monitored shaft revolutions on the discrete base scale, we managed to reproduce exactly the same sequence of indices as the existing application.

We also have to check that the interpolation algorithm and implementation produce the same sets of values.

IV. Online Computation of the Indicators

A. The on-line computation: management of inputs

For the sake of simplicity, we present the design of an on-line version of the health indicator considering the simple case in which there is a single phase sensor (on the tail rotor), and a single accelerometer, on a rotating element (TDS1) with speed ratio 4 (rotating 4 times faster than the tail rotor). Vibration samples are acquired at $15630Hz$. This is enough to show the impact of the connection of the MPPA to the acquisition card, with the samples arriving in packets.

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 4 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 is available for further treatments.

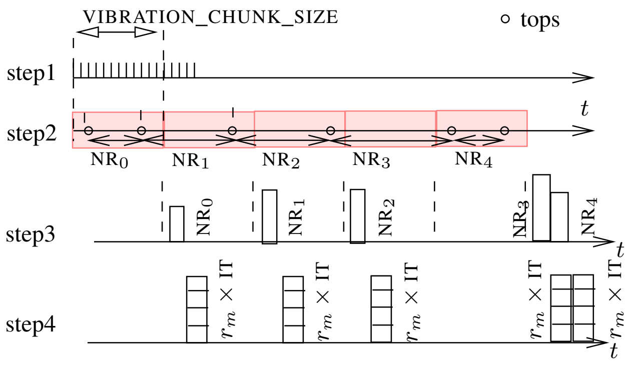

Fig. 4: Preparation of inputs

step1 represents the discrete samples, taken at frequency fs in the range $[1 - 31]kHz$. step2 illustrates the transmission of packets of size VIBRATION_CHUNK_SIZE from the acquisition unit to the MPPA processor. 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.

On step3 we gather samples per revolution (i.e., between 2 tops). The data are available a bit later than the end of the packet. The vertical rectangle is an array of samples of size NR_0, then NR_1, etc. Notice that NR_3 and 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

$$\text{VIBRATION\_CHUNK\_SIZE}/(\frac{\text{fs}}{60 \times v_{\text{ref}}})$$

where fs is the sampling frequency (Hz) and v_{ref} is the rotational speed of the reference shaft (given in revolutions/minute).

step4 shows the interpolation phase: when an array 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 rotation angular velocity of the reference shaft is *constant* during *one* revolution. We now have arrays of $r_m \times \text{IT}$ points. The computation of the indicators starts from there.

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 on-ground computations, the interpolated points are placed according to the global speed average. This is clearly not feasible in the on-line version, where we use a incrementally computed *average until now*, instead.

V. The Kalray MPPA-256 Many-core Processor

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.

Determining precisely the worst-case execution time (WCET) for a many-core architecture is very challenging due to the many contention points due to shared resources. For example, the experiments described in [6] examine the memory access latencies for read and write operations while increasing the number of interfering cores. On the Freescale P4080, the latency of a read (respectively write) operation varies from 41 to 604 cycles (respectively 39 to 1007 cycles) depending on the total number of cores running competing tasks.

Some recent developments in the microprocessor industry address this problem specifically. This is the case of the *Kalray MPPA-256 (Multi-Purpose Processing Array)*. This type of processors 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. According to [2], the benefits of the Kalray platform for critical real-time systems are: deterministic computation, deterministic and predictable response times, low power and high performance.

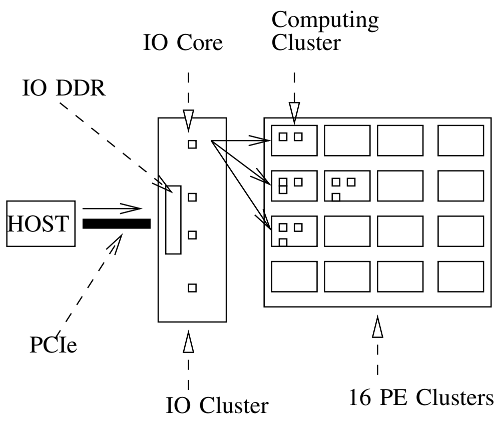

Fig. 5: Abstract view of the MPPA-256 Many-core Architecture (adapted from [1])

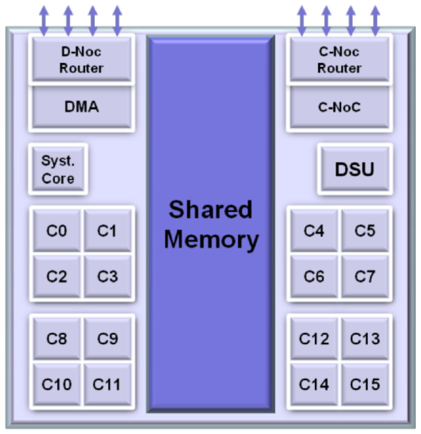

Fig. 6: MPPA-256 Compute Cluster (extracted from [1])

The *MPPA-256* by Kalray is an array of 16 computing clusters and 4 specialized Input/Output (IO) clusters connected by a NoC (Network on Chip). The global architecture of the *MPPA-256* is depicted in figure 5. Each square box corresponds to a computing cluster. The four I/O clusters are represented on the four sides (east, west, north and south). Two of them are connected to a PCIe controller and the two others to an Ethernet controller. Each IO cluster has 4 cores.

A computing cluster is illustrated in figure 6. It is composed of 16 computing cores (each core is called a *Processing Element* or *PE*), one system core (Resource Management), one NoC Rx (receiver) interface, one NoC Tx (transmitter) interface, one DMA (Direct Memory Access) and one DSU (Debug System Unit). They access concurrently the 2MB shared memory. The overall architecture provides separate memory banks, and reservation mechanisms on the NoC, which also contribute to predictability.

VI. MPPA Implementation Experiments

A. Time-stamping tool

For our experiments, we need to gather precise timing data from the MPPA target. The MPPA IO cluster, and its internal clusters, each have an internal clock running at 400Mhz. These counters have the same period, but not the same phase (they are *mesosynchronous*). The maximum observable offset is around 100 cycles. This means that, when taking a time-stamp T_0 in the IO cluster, and a time-stamp T_1 in the internal 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). But we need to perform on-target measurements. Kalray also provides a target trace capability, but the tracing is too intrusive. We designed a lightweight tracing mechanism, by adding time-stamps to the data-flow before/after each data transmission to/from a processing element (PE). For this we need to change the type of the data transmitted.

Instrumentation at the software level always has a impact on the execution. In some cases, it can change the timing significantly and even reorder events. We addressed this problem in the following way: although adding the time-stamps to the data does change the behavior of the system, we observe that the changes are essentially the ones that would be observed with larger data (transmission time, packet storage). The only timing effect that would not be observed with larger data is the time it takes to compute the time-stamps themselves, but this can safely be neglected.

The number of time-stamps that are necessary to perform useful measurements has to be confronted to the cost of transmitting data on the network on chip. First, we choose the *granularity*, i.e., the minimum packet size with which we associate time-stamps to follow the route. The chosen granularity is set to one input sample processed by each core PE. To measure the MPPA latency, the latency of the entire system (MPPA + host), and the computing duration on each worker, we need around 20 time-stamps. Each cluster has a Debug System Unit (DSU) offering a 64-bit counter for time-stamping. We assume that our measures do not exceed 10 seconds. For measuring up to 10s with the 2.5ns MPPA clock period, we need a 32-bit counter. With all our experiments, the time-stamp overhead in the data transmitted is estimated to be around 1%.

B. Dimensioning Experiments

In [3], we show that the communication through the IO cluster becomes the main bottleneck of the system. We implement a pipeline architecture with a double-buffer communication to parallelize the communication and the internal cluster processing. Another case study on the same platform described in [4] had to address the same problem.

In order to observe the end-to-end latency, we made an experiment where data is received by the MPPA, and sent back to the host, without any computation.

The experiment is performed for increasing sizes of data chunks, and 100 times for each size. The MPPA latency remains less than 200 μs for sizes ranging from 4 to about 10000 bytes.

Since the size of one sample data is 4 bytes (a tuple made of a vibration measure and a phase sensor), we can choose any size lower (and close) to $10000/4 = 2500$ for the size of packets to be sent by the HOST to the IO cluster. We actually choose VIBRATION_CHUNK_SIZE $= 2048 = 8192/4$ for the advantage of manipulating power-2 sized arrays.

C. On-line HMS experiment

Once chosen the data chunk size for transmission, we have set up an experiment to mimic the on-line computation of indicators. On the MPPA side, we use 2 threads on the IO cluster core, ioreader and iowriter, and a Worker thread running on a computation core. The ioreader transmits the vibration data coming from the host to the Worker thread that computes the indicators, and iowriter sends back indicators to the host.

On the host side, the host mimics the actual sensors by sending packets of vibration data stored in files. Those data were collected in real operation conditions from a servicing H 175 aircraft, and correspond to the acquisition of 224 monitored shaft revolutions.

1) OM1 on-line indicator results: The OM1 indicator is computed as follows:

$$2 \times \text{Module}(\text{FFT}(\text{AverageSignal}))$$

where AverageSignal represents the average signal of all 224 monitored shaft revolutions in the off-line version of the application. The on-ground OM1 is equal to $1.823g$.

Unlike the on-ground version where only one OM1 is computed by using the average signal of 224 shaft revolutions, the on-board version computes OM1 by using a growing window of samples.

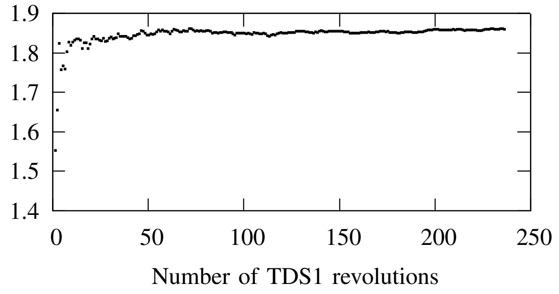

Fig. 7: Evolution of indicator OM1 according to the number of revolutions of TDS1

Online OM1 at step k is calculated by averaging all the $k-1$ previous shaft revolutions. OM1 converges from the 50th shaft revolution ($k = 50$) in Figure 7. When $k = 50$, OM1 is equal to $1.84g$. The relative error of OM1 computed in two different manner (on-ground versus on-board) is equal to 0.9%.

Comparing two OM1 values (on-ground and on-board) does not make sense. People who analyze the indicators results are not interested in absolute values of indicators but in the trend of indicators. In other words, given the same set of samples, the results of indicators given by the two methods must have the same trend.

2) Comparison of on-ground and on-line OM1 indicator: In this experiment, we monitor the equipment *input drive shaft left*, whose rotational speed ratio is $r_t = 16.82$ (it rotates 16.82 times faster than the tail rotor). The purpose of this experiment is to compare the indicator OM1 in two cases. The first case corresponds to the OM1 indicator calculated using the global algorithm (recall, a single value of OM1 is calculated after interpolating and averaging all the monitored shaft revolutions). However, in the case of the incremental algorithm, one OM1 is calculated at each monitored revolution by using a growing window. We compare the last value of the incremental OM1 indicator and the OM1 computed using the global algorithm. The experiment is performed on 20 raw vibration files acquired during the same flight conditions. Figure 8 shows the OM1 indicator computed using different vibration files with the global and incremental algorithms. We note that the error is tolerable in the case of OM1 indicator. For instance, the global 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 the difference is very small. Finally, the maximum relative difference after computing OM1 indicator using 20 raw vibration files is equal to 2.65%.

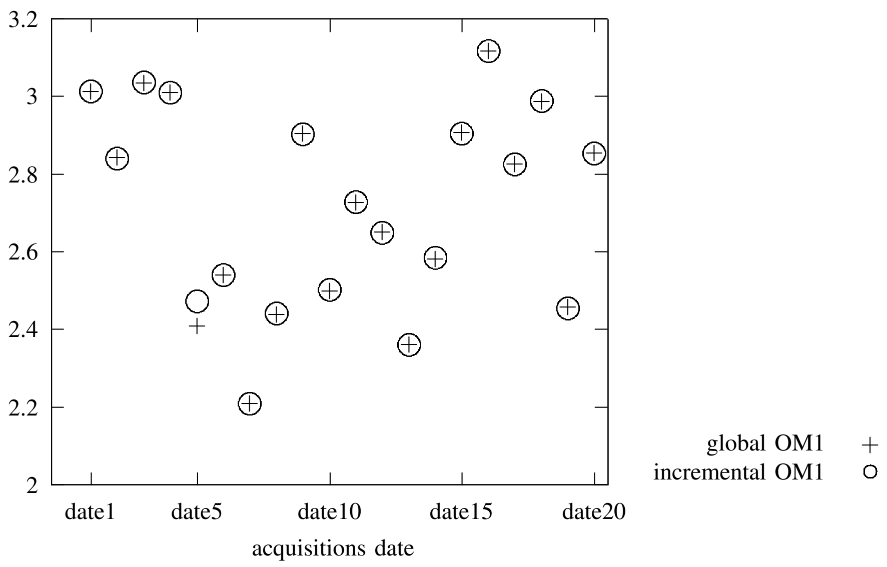

Fig. 8: Comparison of on-ground and on-line OM1 indicator

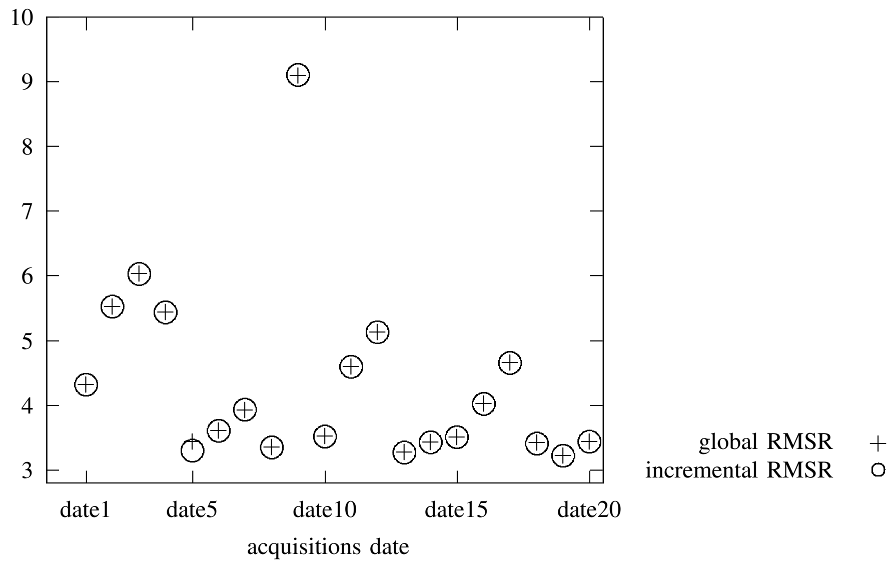

Fig. 9: Comparison of on-ground and on-line RMSR indicator

3) Comparison of on-ground and on-line RMSR indicator: The RMSR (Root Mean Square Residual) indicator is calculated as follows:

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

$AverageSignal$ 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 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 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. This experiment has the same parameters as the previous one (same acquisition file and monitored shaft) and the purpose is to compare the RMSR indicator calculated using the global and the incremental algorithm. We replace the global AverageSignal of the existing HMS by the average signal "*until now*". Figure 9 shows that the difference between the 2 versions is very small. The maximum difference is observed at date5 and equals to 4%.

VII. Conclusion and Future Work

We have transformed an off-line on-ground computation of health indicators, in which the global average of a signal is used, into an on-line embedded real-time computation of the same indicators, in which the average is done on a growing window. We used these experiments in order to validate the error induced on the result of the computation, and to provide dimensioning information for the implementation of the full application.

The lessons learnt are detailed below. First, any globally computed value has to be transformed into some feasible incremental version (the same function computed on either a sliding window, or a growing window). 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. 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.

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.

Second, some tools on the execution platform would really help. The host PC (which is used first, before the actual acquisition card is connected to the MPPA) could be real-time. It would allow more precise estimations of the overall final behavior, without the need for the acquisition card, in early stages of development. A precise timing tracing tool is also needed (either with help from the hardware for real executions, or with a cycle-accurate simulator of the execution platform). We designed our own timestamps mechanism, knowing that, for the application under study, intrusiveness is limited. But this should be studied again for another application, which is not satisfactory.

Further work will be done along several lines. First, we will finish the development and test of the full HMS application on the Kalray MPPA, connected to the real acquisition card; the full HMS involves several sensors, takes input values at various sampling frequencies, and takes into account several distinct speed ratios with respect to the reference shaft. Given the computing power of the MPPA, it may work satisfactorily without optimizations. The next step would be to develop solutions in which the HMS application can share the Kalray with another application, raising questions about the criticality level of several applications sharing the same platform.

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, but the complete procedure has not been conducted in full for now.

A longer-term prospect is to experiment with machine learning techniques applied in the domain of predictive maintenance, and to explore the idea of implementing such a technique on the Kalray MPPA, to be connected to the computation of the indicators. The coupling of these computations would give an on-ground real-time surveillance and maintenance decision.

References

[1] B. de Dinechin, R. Ayrignac, P.-E. Beaucamps, P. Couvert, B. Ganne, P. de Massas, F. Jacquet, S. Jones, N. 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.

[2] B. D. De Dinechin, D. Van Amstel, M. Poulhiès, and G. 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.

[3] M. Lo, N. Valot, F. Maraninchi, and P. Raymond. Implementing a real-time avionic application on a many-core processor. In *42nd European Rotorcraft Forum*, Lille, France, sep 2016.

[4] D. Madroñal, R. Lazcano, R. Salvador, H. Fabelo, S. Ortega, G. Callico, E. Juarez, and C. Sanz. Svm-based real-time hyperspectral image classifier on a manycore architecture. *Journal of Systems Architecture*, 80:30–40, 2017.

[5] Mathworks. *Predictive Maintenance with MATLAB; Avoid costly equipment failures by using sensor data analytics.* Mathworks, 2016.

[6] J. Nowotsch, M. Paulitsch, D. Bühler, H. Theiling, S. Wegener, and M. 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] Sikorsky. Sikorsky, phi, metro demo real-time hums on s-92. 3 2017.