HAL
archives-ouvertes.fr

IMPLEMENTING A REAL-TIME AVIONIC APPLICATION ON A MANY-CORE PROCESSOR

Moustapha Lo, Nicolas Valot, Florence Maraninchi, Pascal Raymond

▶ **To cite this version:**

Moustapha Lo, Nicolas Valot, Florence Maraninchi, Pascal Raymond. IMPLEMENTING A REAL-TIME AVIONIC APPLICATION ON A MANY-CORE PROCESSOR. 42nd European Rotorcraft Forum (ERF), Sep 2016, Lille, France. 42nd European Rotorcraft Forum (ERF), 2016. <hal-01718139>

HAL Id: hal-01718139
https://hal.archives-ouvertes.fr/hal-01718139

Submitted on 27 Feb 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

IMPLEMENTING A REAL-TIME AVIONIC APPLICATION ON A MANY-CORE PROCESSOR

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

A recent microprocessor architecture breakthrough provides a many-core processor that offers timing guarantees. It gives us an opportunity to study its applicability to avionics systems. We select an avionics function that requires both high processing power and some response time guarantees. The Helicopters Health Monitoring System (HMS) performs signal processing on vibration data, to raise some alerts for the operating crew. The computation requires a high processing bandwidth and the alerting requires a bounded response time. These characteristics makes the HMS a good candidate for an experiment in implementing avionics functions on a many-core processor.

I. Introduction

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 processors are connected on chip through a bus, sharing the same memory. Many-core processors are usually structured into two layers: processors 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). In both multi-core and many-core processors, the potential interferences induced by the shared memory, the shared buses, the NoC, etc., are bad for predictability and response-time guarantees.

A recent development in the microprocessor industry addressed this problem. The *MPPA-256* by Kalray (MPPA stands for "Multi-Purpose Processing Array") has been designed taking into account determinism and response-time requirements. Each core is a simple processor, allowing for good execution-time predictability. The overall architecture provides separate memory banks, or reservation mechanisms on the NoC, which also contribute to predictability. According to [2], 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 outcome of this case study shall provide some performance capability on the MPPA target with the following variation points: the sampling frequency is set by configuration to a value in the range of 1..25 KHz. The number of sensors is in the range of 1..256.

We first introduce the many-core architecture and the HMS. We then explain the constraints and main ideas for an implementation of the HMS on the many-core architecture, exploiting its computing power and offering good response time guarantees.

II. 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 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 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.

This publication and the related work was performed in the scope of the CAPACITES research project, supported by the French authorities through the "Investissements d'Avenir" program

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 off-line, and when the helicopter is on ground. This architecture requires a huge storage capacity, and a large network bandwidth for data offloading.

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. Because of the avionics constraints, this processor should also provide low power and predictable response times. We study the implementation of the HMS on the Kalray MPPA-256 processor.

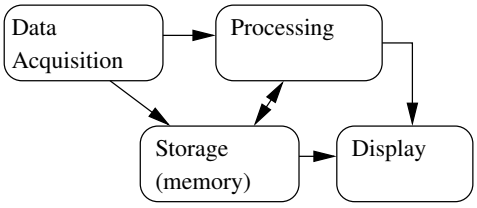

Figure 1: Functional Architecture of the HMS

The current functional architecture is described in Figure 1. The *Display* will not be studied here. 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.

III. A Real-time avionic application

Some unpredictable events might occur upon undetected mechanical part failure. This case study is intended to evaluate a computing platform and a software architecture that provide real time detection indicators, which could be used by crew. Such a system would require several enablers:

- real time computation
- specific indicators and sensors. Current on-ground indicators compute trends across several flight cycles. A real time indicator should detect a rapid change in the dynamic envelope.
- accurate indicators (no false alarm)

The case study will focus on the real time computation capability. The other enablers are not in the scope of the study.

The HMS system requirements for the MPPA processor shall address a range of 1 to 256 accelerometer sensors. To evaluate the platform performance capabilities on the HMS system, we will implement the most performance-demanding sensor indicators. The main gear box bearing parts correspond to the highest rotation frequency. Fatigue occurring on the inner or outer race induces some spike any time a ball crosses the race defect. According to [1], this spike period is the bearing period divided by the number of balls in the bearing. Therefore, the bearing sensor indicator requires the highest sampling frequency to extract spikes high-frequency harmonics. Actually, the signal processing channel for piezzo accelerometer sensors can reach up to 20KHz. For this case study, we will define a sampling frequency range of 1 to 25 KHz. The bearing sensor workload requires to compute an envelope FFT. The envelope itself is a Hilbert transform composed of one FFT and a reverse FFT. We can add a window to tune the spectrum leakage effect. The window shall be carefully chosen to detect transient spikes. To compute the spectrum, there are mainly two strategies:

- Use a magnetic sensor to identify the number of samples in a single bearing period (which might vary over time)
- Perform a sliding analysis with a constant number of samples

The first solution requires to compute twiddle factors before each FFT computation which is very inefficient. It might be insteresting to evaluate harmonics of the bearing period, but has no advantage to track spikes occurred by balls rolling on a race. The second solution enables efficient computation, and provides a constant resolution by design.

In this study, we will compute 1024-plot FFTs, which will provide around 25Hz resolution @ 25KHz and 10 Hz resolution @ 10 KHz. The resolution might be increased with a logarithmic increase of CPU demand.

In the sequel, we will use "Log" to denote base-2 logarithm. A naive FFT complexity is $N * Log(N)$. Therefore increasing by 4 the resolution leads to a complexity increase of $4 * N * log(4 * N)/(N * log(N)) = (4 * (log(4)) + 4 * log(N))/log(N) = 8/log(N) + 4$. When increasing resolution by a factor of 4, we are also computing 4 times more samples. Therefore, the complexity by sample is $(8/log(N) + 4)/4 = 2/log(N) + 1$. For $N = 1024$, the complexity increase to raise resolution to 4096 would be $2/6 + 1 = 1,33$. This factor does not

take into account the memory locality penality to process 4 times more data.

IV. Static mapping on the MPPA-256

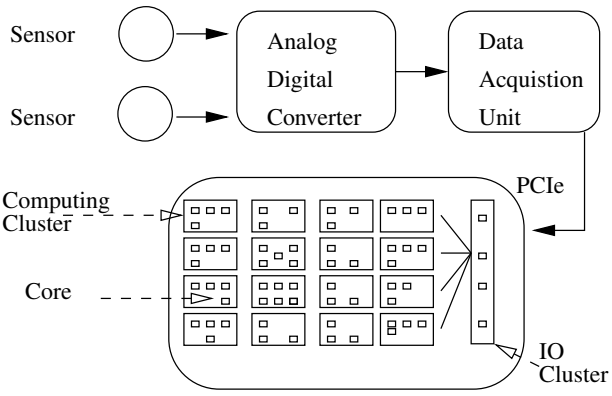

Figure 2: Mapping on the MPPA-256

Only the processing part of Figure 1 is mapped on the MPPA-256. The resulting architecture is described by Figure 2. The sensors and the associated analog-digital converters are connected to a *data acquisition unit*, which sends digital formatted data to the MPPA-256, through a PCI bus. A similar structure would be needed for the output of the results to some embedded equipment.

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.

The MPPA 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 and each sensor delivers N samples. 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 3 shows the distribution of sensors across IO cores in this case.

However, the current MPPA 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 improve timing. Figure 4 shows the limitation with the current MPPA implementation.

In this static assignment, our code would decide before execution which processors manage which tasks, which is sometimes referred to as *bounded multi-processing (BMP)*. One objective of the mapping will be to minimize end-to-end application execution time, i.e., the time it takes for one sample

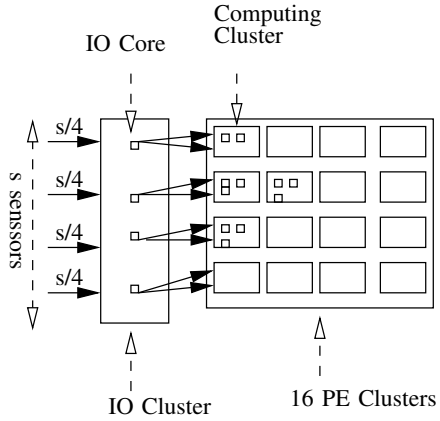

Figure 3: Distributing Sensor Samples on the cores of the IO Cluster

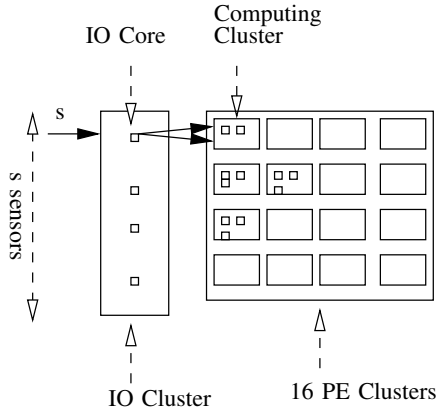

Figure 4: Using only one core of the IO cluster for the Sensor Samples

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 5: there are three steps, that have to be performed in sequence because each part depends 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.

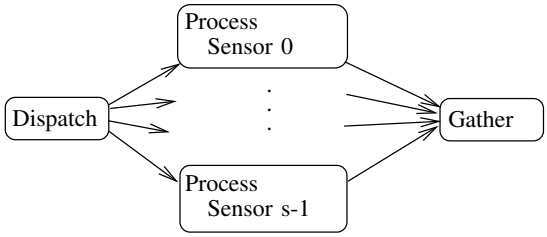

Figure 5: Functional structure

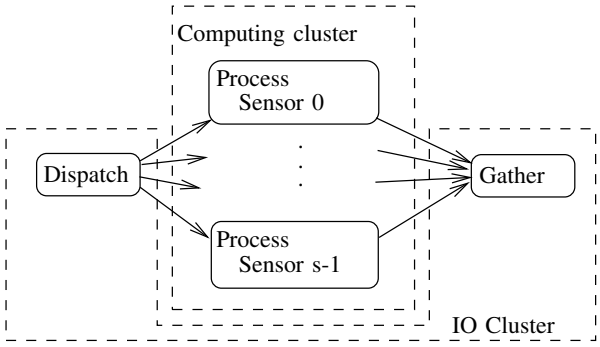

Figure 6: Dispatching and Gathering mapped on IO Cluster

A. Dispatching *and* Gathering *in the IO cluster*

Figure 6 illustrates this choice. In this configuration, the IO cluster manages both *Dispatching* and *Gathering*, implemented as two tasks running on the same processor. *Processing* the data for each sensor is allocated to one of the PEs of a computing cluster.

The MPPA 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 the sending thread. Obviously, the software overhead to call the send or receive packet services and use 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.

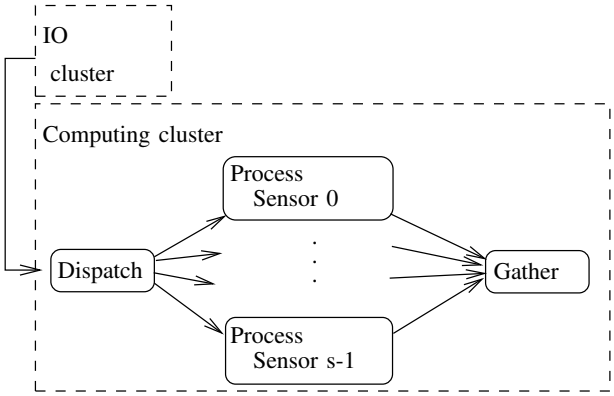

Figure 7: Dispatching and Gathering mapped on the computing cluster

B. Dispatching *and* Gathering *in the computing cluster*

Figure 7 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 assignement i.e., to keep all processors busy as much as possible and avoid overloading of any single resource (NoC route, DMA, processor). We need the next data to be processed to be available at the moment when processors work on the current one. Assume we have s sensors, a algorithms to compute (the details of "processing" sensor x) and p processors. We have two choices to distribute calculations over processors. We describe each choice in details below.

1) Parallelizing the algorithms: It means dedicating one processor among the p processors to each particular algorithm among the a, to compute data coming from all sensors. The algorithms involved in the HMS function exchange data: outputs produced by one of them are often re-used by another. If algorithms are allocated to distinct processors, this involves a synchronization overhead, which depends on the number a of algorithms; the communication overhead will also be significant.

2) Parallelizing the sensor data: It means allocating one processor among the p processors to one sensor among the s sensors, and to compute all a algorithms for the same sensor on that processor. Sensors are functionally independent, thus threads can run without needing any data exchange. The *load balancing* seems to be perfect, since each processor computes several algorithms sequentially on one sensor data. The processors do not need to

communicate. However, the processors 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 processors. 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 7). In one computing cluster, the maximum number of processors usable to compute algorithms is therefore 14, among the 16 processors that are physically avalaible. We will call these processors *workers* in the sequel.

V. Experiments

A. Timestamping tools

For our experiments, we will need to gather some timing data from the MPPA target. The MPPA IO cluster, and its internal clusters, each have an internal clock running at 400Mhz. These counters are synchronous but their initialization is not. The offset is around 100 cycles. This means that, when taking a timestamp T_0 in the IO cluster, and a timestamp 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 intrusive and we want an agnostic platform trace capability. We designed a tracing mechanism, by adding timestamps to the data-flow before/after each data transmission on to/from a processing element (PE). For this we need to change the type of the data transmitted. However, adding the timestamps does not change our application functionally. Figure 8 gives an example of changing the data type coming from the host processor.

```
  // 32 bits timestamp
typedef unsigned long timestamp;

// input timestamps
typedef struct HDRin {
    // HOST Writer T0
    timestamp HOSTWRT0;

    // IO Writer T0
    timestamp ioWRT0;

   // IO Writer T1
    timestamp ioWRT1;

    // PE reader T0
    timestamp  peRDT0;
}HDRin;

typedef struct dataIn{
  HDRin hdrin;

 // input samples vector
  float InputSamples[VECTOR_LENGTH];
}dataIn;
```

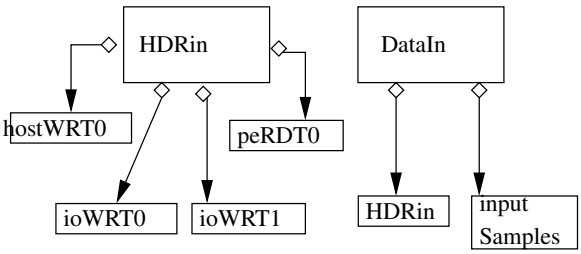

Figure 8: HOST/IO senders data type

The number of timestamps 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 timestamps to follow the route. The chosen granularity is set to one input sample processed by each worker 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 timestamps. 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 these figures, the timestamp overhead in the data transmitted is estimated at 1% for a 1024-plot FFT sample.

B. Description of the Experiments

In all our experiments, we use only one processing cluster among the 16 physically available. We will first evaluate a synchronous dataflow architecture, in order to observe end-to-end data latency.

Synchronous here means that, on the host processor, we wait until a complete treatment of a set of samples has been performed, before sending a new set of samples.

Then, we will evaluate a pipelined architecture to improve throughput. In this architecture, the host processor send samples as fast as possible. At each pipeline stage, we are waiting for the availability of the previous stage output, and the availability of the next stage input storage.

C. Synchronous dataflow architecture

The synchronous dataflow architecture consists of: a HOST thread that sends samples to the IO 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, which finally transfers it to the HOST. The host thread must receive data before it sends another sample set. We measure the MPPA latency: the time it takes for one bit to make a complete round trip through the IO and the internal clusters.

Figure 9 shows the MPPA latency for various data sizes. For a given packet of samples, the latency is measured 100 times. It is less than 200 microseconds between 1 and 8192 bytes. It means the latency is almost the same when sending a small data packet, for instance 4 bytes or 8192 bytes. This result permits us to choose 8192 bytes as the size of the smallest packet sent by the HOST to the IO cluster.

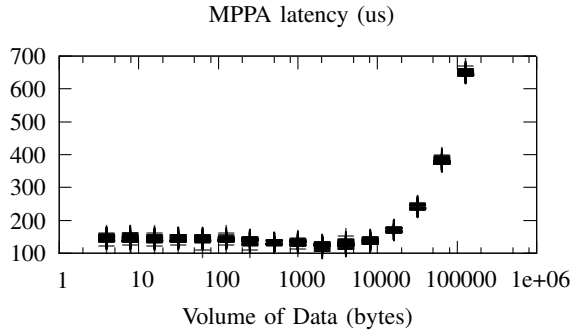

Figure 9: MPPA Latency

This synchronous architecture has two main drawbacks:

- The throughput depends on the IO latency
- It is impossible to pipeline the HOST, the IO, and the Internal Cluster computations.

D. Pipelined architecture for maximal throughput

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.

2) Data Architecture: In order to avoid the problems encountered with the abovementioned *synchronous dataflow* architecture, we separate *sending* and *receiving*, allocating them to different threads. The PE reader thread (See Figure 10) reads data of type *vector [N][S]* (recall the HMS function has S sensors and each sensor delivers N samples) 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 12).

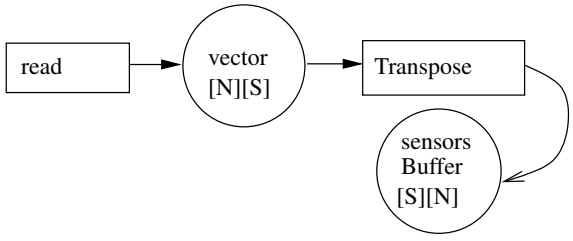

Figure 10: PE Reader Thread

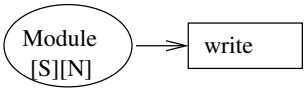

Figure 11: PE Writer Thread

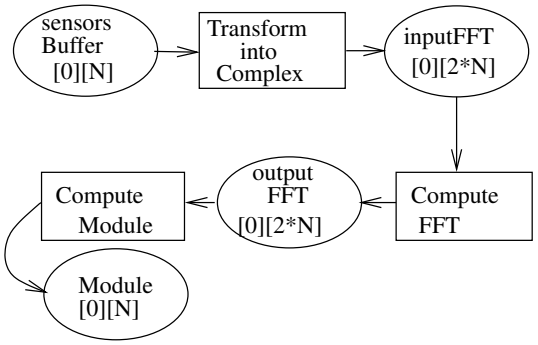

Figure 12: Worker0 Dataflow

`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 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 12: 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.

3) Control Architecture: The structure is the following. We use two threads on the host: hostwriter and hostreader. We also use 2 threads on the IO cluster core: iowriter and ioreader. 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 hostwriter sends packets of 8192 samples grouped in the `DataIn` structure to thread iowriter with the hostWRT0 timestamp (see Figure 8). This timestamp indicates the beginning of data transmission. This data exchange is done through a Pcie buffer. Reading and writing are performed by Posix functions. These functions are blocking, so no synchronization is needed between the host processor and the IO cluster.

The thread hostreader receives samples from ioreader. They represent the computation results of all workers, associated with all timestamps taken along the route. Then it set its timestamp hostRDT0 after reading. hostRDT0-hostWRT0 represents the overall latency of the system (MPPA + HOST). This measurement is only relevant for a real-time HOST.

The thread iowriter reads the `DataIn` structure and positions its ioWRT0 timestamp. Then it transmits it to the thread PE reader before positioning its second timestamp ioWRT1. The thread ioreader reads the data structure coming from the PE Writer thread and sets its 2 timestamps.

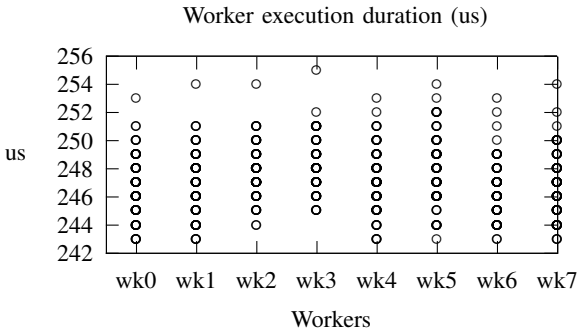

Figure 13: Worker Execution Duration after computing 1 FFT and 1 module in 100 host loops; Host configured in best effort

4) Results: 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.

a) Computing 1FFT and 1 Module: First we measure the processing time of each worker (see

Figure 13). This duration is between 242 and 256 microseconds. This means a jitter of 5.4%. We have the same results from 100 to 1000 loops.

On Figure 14, 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 842 μs. This gives a duration of treatment of 242 μs. This value is consistent with Figure 13. 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 $600/250 = 2.4$. To benefit from the computation capabilities offered by the MPPA, 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 15).

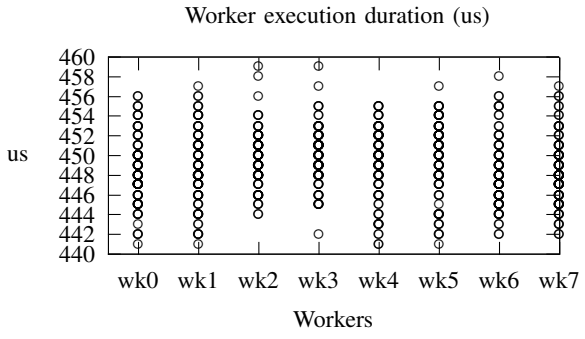

Figure 15: worker Execution Duration after computing 2 FFTs and 1 module in 100 host loops; Host configured in best effort

b) Computing 2 FFTs and 1 Module: We repeat 100 times this same experiment. Each of the 8 workers takes between 440 and 460 μs to compute 2 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 842 μs is now at 1040 μs. The workers wait only 200 μs (between 1000 et 1200 μs) instead of 400 (see Figure 16). The ratio Data Transfert Duration/Processing Duration was equal to 2.4 and now becomes 1.3.

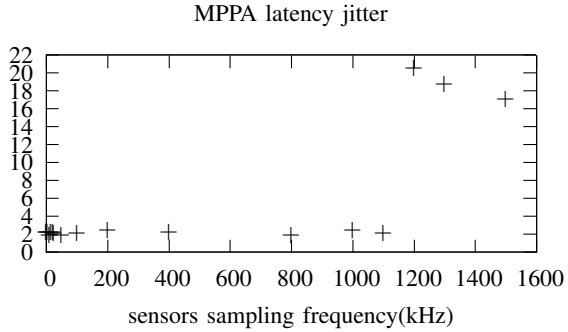

Figure 17: MPPA latency jitter measured with various sensor frequencies

c) MPPA latency jitter: 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.

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

d) Pipelined architecture, driven by the bearing frequency: In this last experiment, we choose a particular sensor frequency: 15kHz (the bearing sampling frequency). It corresponds to $Te_{1024} = \frac{1024}{15kHz} = 68.26ms$. 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 18. It shows that between 2 HOST emissions, each worker can compute for one sensor

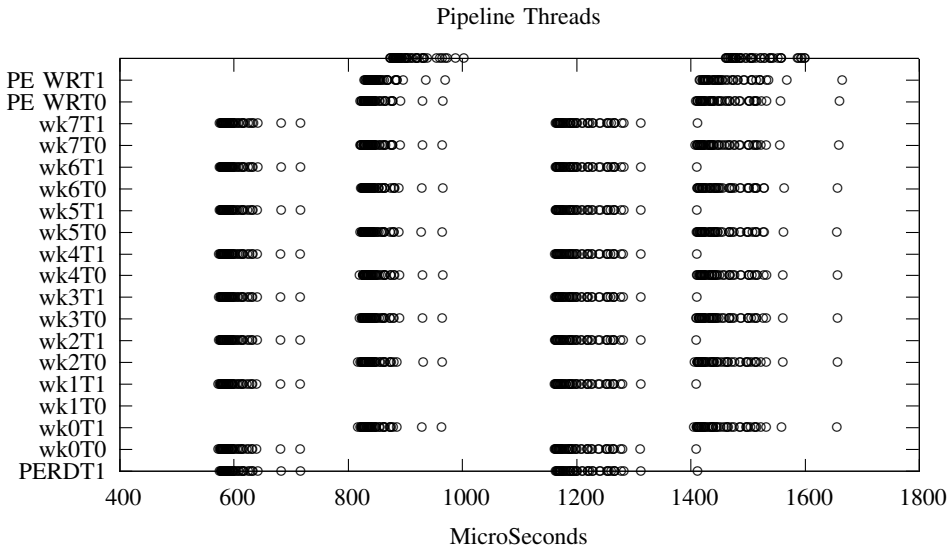

Figure 14: 8 Pipelined workers compute 1FFT and 1Module in 100 host loops; Host configured in best effort

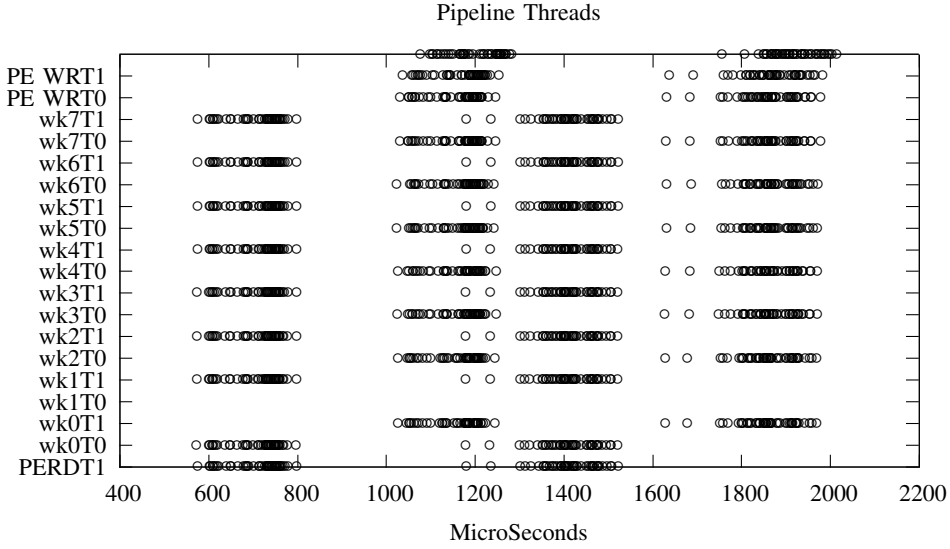

Figure 16: 8 Pipelined workers compute 2FFT and 1Module in 100 host loops; Host configured in best effort

the equivalent of $Te_{1024}/250 = 273$ FFTs or 1 FFT for 273 sensors.

VI. Conclusion

With our choice of mapping, one MPPA cluster of the MPPA processor owns up to 14 workers. Taking into account the parameter ranges mentioned in the introduction (Fe $\leq$ 25 KHz and number of sensors $\leq$ 256), one MPPA cluster is able to compute a workload of $(((1024samples/25KHz)/250\mu s) * 14workers)/256sensors = 8,96$ 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 provide 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.

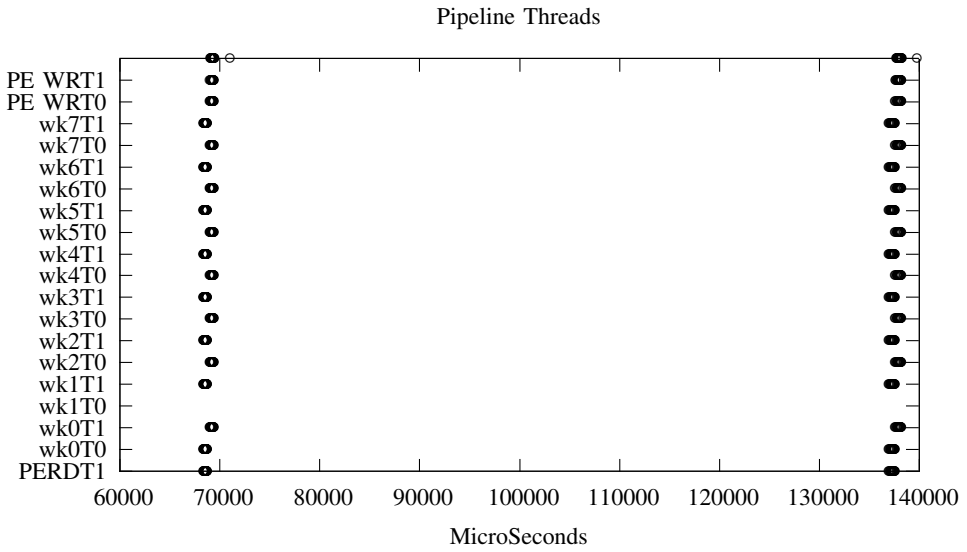

Figure 18: Workers pipelined - 3FFT and 1Module - Periodic Host - 100 host loops

Future work will concentrate on legacy indicator implementation, and new indicator specification (threshold learning, alarm detection logic). It will also measure the execution time impact of several cluster traffic on the NoC.

REFERENCES

[1] P. Arques. *Diagnostic predictif et defaillances des machines, Theorie-Traitement-Analyse-Reconnaissance-Prediction*. Editions TECHNIP, 2009.

[2] 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.