Master of Science in Informatics at Grenoble
Master Mathématiques Informatique - spécialité Informatique
Option Parallel, Distributed and Embedded Systems

Implementing critical systems on many-core architectures : Towards solutions preserving determinism

Hanan KANSO

23rd of June 2014

Research project performed at Verimag

Under the supervision of:
Prof. Florence MARANINCHI and Prof. Matthieu MOY

Defended before a jury composed of:
Prof. Noel De-PALMA and Prof. Martin HEUSSE
Prof. Nabil LAYAIDA and Prof. Alain GIRAULT
Prof. Olivier Gruber and Prof. Arnaud Legrand

This work has been partially supported by the LabEx PERSYVAL-Lab (ANR-11-LABX-0025)

Acknowledgements

I have taken efforts in this project. However, it would not have been possible without the kind support and help of many individuals. I would like to extend my sincere thanks to all of them.

I am highly indebted to Verimag's team for welcoming me in their laboratory and for their kind guidance during the internship. I would like especially to thank my supervisors Prof. Florence MARANINCHI and Prof. Matthieu MOY for accompanying me during my first research experience. I would like to express my gratitude towards my master's tutor Mr. Christophe RIPPERT for his co-operation and encouragement and towards the PERSYVAL association who supported this project. I would like to express my deepest appreciation for jury members for giving me time and attention. On the personal side, I would like to express my special gratitude for my parents for being always beside me and for my special person, my fiance, for his endless love and support. My thanks and appreciations also go to my family's members and friends.

Abstract

Shifting to many-core architectures is becoming compulsory for real-time applications demanding intensive computing capabilities. Nonetheless, the gain in performance of many-core architectures does not come for free. In particular, it comes at the cost of predictability and determinism issues. Logical determinism is the property of programs that for the same sequence of inputs always give the same sequence of outputs. This property is essential for critical systems which are systems whose failures can result in people deaths or injuries, in severe damage to equipments or to the environment. Although many-core architectures do not all respect criticality constraints, some designers have taken into account this problem as for example the Kalray MPPA-256 processor. These architectures appear as good candidates for implementing critical systems. Based on that, we develop a solution that will be able to enforce logical determinism by construction.

Contents

Abstract **ii**

1 Introduction **1**

2 Reactive Systems and their implementation **5**
2.1 Reactive Systems . . . 5
2.2 Programming languages and the notion of task . . . 6
2.3 Single-task implementation . . . 8
2.4 Multi-task implementation . . . 10

3 From Single to Many-core architectures **13**
3.1 Single-Core architectures . . . 13
3.2 Multi-Core architectures . . . 13
3.3 Many-Core architectures . . . 14

4 Mapping high-level models (Lustre) on many-core architectures
Semantics preservation problem **21**
4.1 The mapping problem . . . 21
4.2 Example . . . 21

5 A deterministic mechanism for the communication via the NoC **25**
5.1 The case-study . . . 25
5.2 Previous work: Semantics-Preserving Multi-Task Implementation . . . 27
5.3 Contribution of the report . . . 28

6 Implementing the solution on the Kalray MPPA-256 **31**
6.1 NoC connectors for the Kalray MPPA-256 . . . 31
6.2 Choice of a connector . . . 32
6.3 Implementing the writer and the reader . . . 35

7 Evaluation **37**
7.1 Example . . . 37
7.2 Instrumentation . . . 37
7.3 Observations . . . 39
7.4 Conclusion . . . 41

8 State of the art **43**
8.1 Non-critical systems 43
8.2 Critical Systems 44

9 Perspectives **47**

10 Summary and Conclusions **49**

Bibliography **51**

— 1 —
Introduction

Critical systems or life-critical systems are systems whose dysfunction can cause people deaths. These systems are ubiquitous. They are present in the infrastructure, in medicine, in nuclear power plants and in many transport domains like avionics and automotive. Safety is the main criteria for these systems. People and institutions have gathered some obligations that a critical system should respect. For example, in the avionics domain, the organizations dealing with avionics safety have specified some development constraints. Two sides were taken into account, software and hardware. On the software side, most of the airborne systems apply the airborne considerations and equipment certification dealt with the DO-178 B document. This document gives guidance on development considerations and specifies criteria on system reliability. Because failures can have different effects, they were classified into different categories depending on the severity of the problem they can cause. Some failures can have no effect on the safety of the flight, like the dysfunction of the video system, when others can have catastrophic effects like a crash.

To guarantee the safety of critical systems, at least preliminary tests should be repeatable; for the same sequence of inputs we obtain the same sequence of outputs. This property is called logical determinism. We aim to have such systems that have predictable behavior. Determinism and performance are often opposed.

In non-critical systems like video games, performance is more likely to be searched than predictability. The limitation of single-core architectures to offer high speed has pushed to shift to many-core architectures. These cores can be homogeneous or heterogeneous. They can communicate via shared memory, buses or network on chips. Nonetheless these architectures come with more complications. These complications can be the cause of the violation of determinism qualification as we will see later. Yet, in critical systems it is not accepted to have better performance at the cost of having less predictable systems. This topic is attracting researchers nowadays especially with the growing interest in high performance systems like many-core architectures.

Some critical systems show a growing need in computation capacity thus the idea of adopting many-core architectures seems inevitable. It is then becoming clear that a problem is to benefit from the power of many-cores without losing the main requirement of critical systems which is the determinism. We mention for example systems in aeronautics, transport, industrial

automation and spaceflight.

The project we are working on during this internship aims to map high-level data-flow programs (Lustre, SCADE [9]) on many-core architectures. This mapping should preserve the semantics of the original program. The semantics of the program are determined by its expected functional behavior. In this context, given a sequence on inputs the original program and its implementation on the many-core architecture should give the same sequence of outputs. The implementation of the program goes through different steps. The high-level model is partitioned into several units of computation called tasks. These tasks will be mapped on the cores of the processor. The mapping has a significant effect on the execution of the program. The scope of this internship is to study, once having a particular partitioning and a particular mapping, what is their effect on the implementation and what can go wrong and violate the original semantics of the program.

The applications we are studying are real-time applications. Real-time applications communicate with the environment by reading data, processing them and writing outputs as fast as needed regarding the speed imposed by the environment. Such an application is described as a set of tasks communicating with each other and having strict timing constraints known as deadlines. The transformation of a high-level description into a set of tasks mapped on the hardware should preserve the semantics of the high-level program as explained above. The hope of reaching this goal would be lost if we were constrained to general-purpose architectures like Intel X86 or X86-64 for example. These architectures and many others focus mainly on optimizations methods to reach high performance. These mechanisms are essentially non-predictable and aim to reach best effort execution time. There are essentially two approaches:

- One approach is to redesign completely the hardware. All the hardware components are replaced with deterministic hardware. All of the optimization mechanisms are removed to eliminate all the sources of non-determinism. This solution is expensive and these architectures are not manufactured by any company.
- The other one has taken into account the problem and made some efforts to eliminate some sources of non-determinism. One of these examples is the Kalray processor MPPA-256. The Kalray MPPA-256 processor was designed essentially to meet the high performance computing needs and low power consumption and to be applicable to critical systems.

We adopt the second solution. We aim to build functional determinism over a non-deterministic hardware. We use the Kalray MPPA-256 architecture that is temporally predictable. Even though, this doesn't imply a deterministic functional behavior. For example, the duration of communications via the NoC on the platform is bounded. Nonetheless, for the same communicated sequence of inputs we do not necessarily obtain the same sequence of outputs. We will discuss later this problem in details.

The main contributions of the project are:

- The diagnosis of logical determinism on many-core architecture (the Kalray MPPA-256)

- A solution to a sub-problem (NoCs) and its implementation on the Kalray MPPA-256.

In the second chapter, we present reactive and real-time systems we work on. The third chapter presents the target platform (many-core architectures). The forth chapter explains why the implementation on many-core architectures may not have the same semantics as the high-level description of the program. In chapter 5 and 6, we present the proposed solution to enforce the preservation of semantics between the description and the implementation. Chapter 7 discusses the consequence on preserving semantics after implementing the solution that enforces determinism. Then, we will talk about other works in the same domain in chapter 8. We finally present the perspectives of the internship and how to use its results to attain the goal of the project before concluding.

— 2 —
Reactive Systems and their implementation

2.1 Reactive Systems

Reactive systems are systems that react continuously to the environment. Figure 2.1 illustrates a reactive system in the environment:

- A calculator is the engine which computes some outputs using some received inputs. The duration of the computation can vary.
- The environment is the element that provides information to the calculator. It also receives outputs from the calculator.
- Components that link both the calculator and the environment: sensors and actuators.

The loop implies that outputs of the calculator usually have effects on next inputs. A simple program that expresses this behavior:

```
while(1){
      read_inputs(I);
      compute_outputs(I,O);
      write_outputs(O);
}
```

Calculators should be able to compute fast enough to be able to capture the next input. The requirement for real-time systems is that the execution time (body loop time) is less than the sampling frequency imposed by the environment. Let's take the example of the temperature controller. To be able to maintain a constant temperature in a room, the controller is fed with the current temperature. If the temperature is higher than the desired temperature, the systems decides to cool the room. Otherwise it should heat the room (see occurrence of inputs and outputs in Figure 2.2) . Control engineers know how to choose the appropriate period of inputs sampling. The period should be small enough to allow the controller to maintain the temperature of the room but not too small to avoid computing unnecessary inputs. Real-time systems should respect the following requirement:

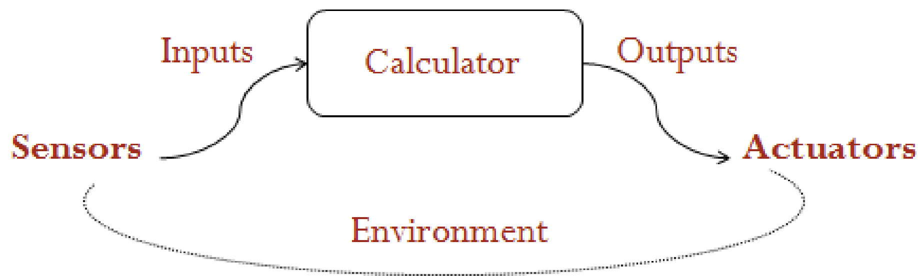


Figure 2.1: Reactive system

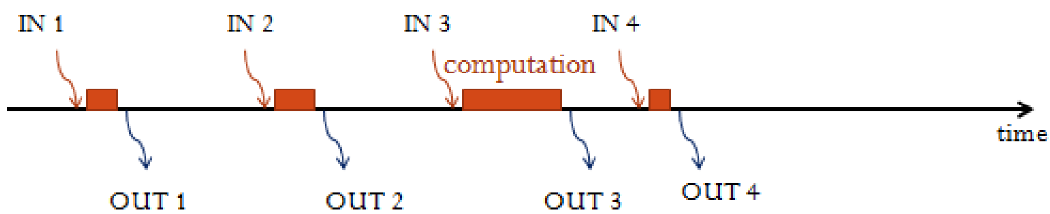


Figure 2.2: Real-time System

reading inputs + computing outputs + writing outputs duration < period

Additional constraints can be added to force writing outputs before the end of the period. This time is called the deadline. Hence, the requirement becomes:

reading inputs + computing outputs + writing outputs duration < deadline

2.2 Programming languages and the notion of task

Applications are described with a high-level model offered by synchronous data-flow languages like Lustre [13]. Lustre is used for critical control software in aircraft, helicopters and nuclear power plants. In Lustre, a variable is an infinite flow of a given type like integers or booleans. A flow possesses also a clock that represents a number of times. At the nth clock's tick, the nth value of the flow is used (discrete time). A program is represented by a network of operators (nodes) connected by wires. Lustre is a declarative languages, thus programs are a set of equations and not a sequence of assignments (no order). Let's look at a simple example. Figure 2.3 shows a simple program that computes the integrator of a flow (in). The related program is:

```
node Integrator (in:int) returns (out:int);
let
    out = 0 -> pre(out) + in;
tel
```

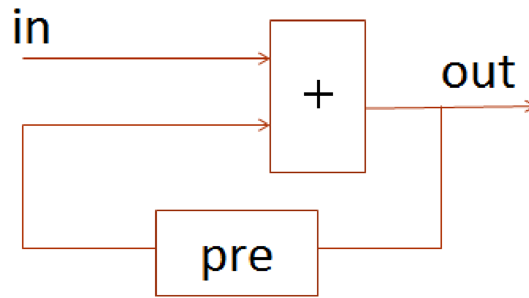


Figure 2.3: Integrator example

Table 2.1: Sampling operator

X	5	7	8	-1	2	3
C	true	false	false	true	false	false
X when C	5			-1		

Table 2.2: Current operator

X	5	7	8	-1	2	3
C	true	false	false	true	false	false
X when C	5			-1		
Current(X when C)	5	5	5	-1	-1	-1

The output flow is initialized by 0 using the initialization operator $\rightarrow$. The operator **pre** is a delay operator. Given a variable it delays the flow by one clock cycle. if X = 0,1,2,3,4... then pre(X) = not defined,0,1,2,3... **Out** accumulates the values of **in**. The main clock in Lustre program is the basic clock. It is possible to define new clocks if we want to express that parts of a program computes more often than others. The **when** operator allows to enable data sampling. Table 2.1 shows a variable X that we want to sample by the clock C. We are interested by reading one input over three inputs. Thus, the new generated variable has one defined value over three regarding the basic clock of X. The dual of this operator is **Current** that redefines non-existing value of a given variable regarding another clock. Table 2.2 shows how to re-define the values of **X when C** with respect to the basic clock.

Programs are usually more complex than that and contain a larger number of nodes. To be handled, they are usually grouped into parts or units of computation. This unit of computation is called a task. A task is a mission that reads some inputs, does some computations and writes some outputs repeatedly. A task has a release time. It is the time when the task becomes able to be executed. If the task is periodic, then it has a period by which it repeats. The periods of different tasks maybe different because the period is related to the sampling frequency of the inputs. When the task is scheduled, it begins to execute. In real-time systems, a task usually

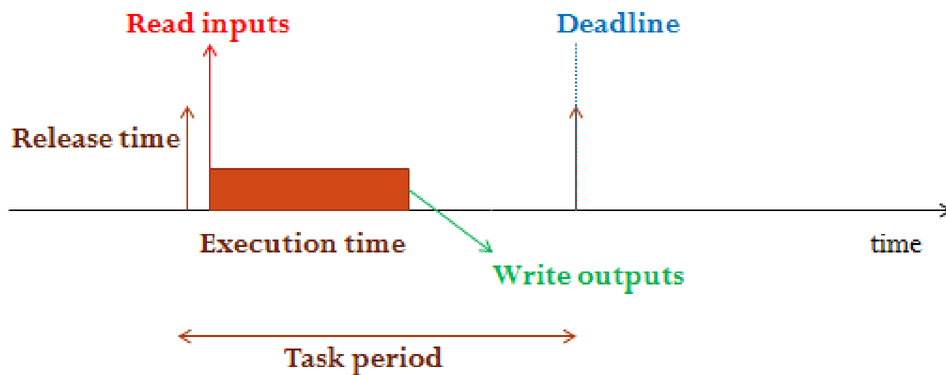


Figure 2.4: Single-task implementation

begins by reading inputs on the sensors. The time it takes to generates outputs is the execution time. It ends by writing outputs on the actuators. A task has also a deadline to finish executing. It should finish computation and write outputs before the deadline. **To simplify we consider that the deadline of a task is its next release.**

2.3 Single-task implementation

A real-time system can be implemented as a single-task (task of the OS). To be clear, there are two different tasks the task of the synchronous program and the task of the OS. The single-task implementation involves reading inputs, a transition function that for some determined inputs generates determined outputs and writing these outputs. The implementation is:

```
Each period do
     read_inputs(I);
     compute_outputs(I,O);
     write_outputs(O);
end
```

Figure 2.4 shows the representation of a single-task implementation in the time. The maximum execution time of the task should be less than the period of the task. This time is called the worst case execution time (WCET). It is the maximum length of time the task could take to execute on a specific hardware platform. In addition, when communication times (reading inputs, writing outputs) take significant time then

the execution time of the task + communication time should be $<$ period of the task.

The single-task implementation is simple and well defined.

Nonetheless, the single-task implementation is limited. We will illustrate this disadvantage with an example. We will take two tasks (synchronous programs tasks) with two different

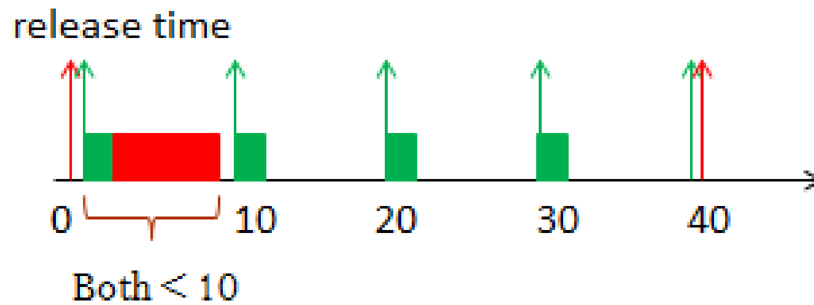


Figure 2.5: Single-task limitation

periods. As said before, the period is related to the sampling frequency of inputs. The task 1 has a period of 40 units of time and has a long execution time. The task 2 has a period of 10 units of time and has a short execution time. The two tasks of the synchronous program can be implemented as a single-task of the OS. If the task 1 accepts to be executed more often the code could be:

```
Every period = 10 do
     read_inputs(IN1,IN2);
     compute_outputs(IN1,OUT1,IN2,OUT2);
     write_outputs(OUT1,OUT2);
end
```

If it is not possible to execute the task 1 more often, it could be executed only one time over the four executions of the fastest task, task 2.

```
Every period = 10 do
     Every 1/4 times do
          read_inputs(IN1);
          compute_outputs(IN1,OUT1);
          write_outputs(OUT1);
     end;
     read_inputs(IN2);
     compute_outputs(IN2,OUT2);
     write_outputs(OUT2);
end
```

The slower task 1 has a longer computation time. Hence, to be able to compute the first and the second tasks, the machine should be fast enough. Figure 2.5 shows the limitation of this

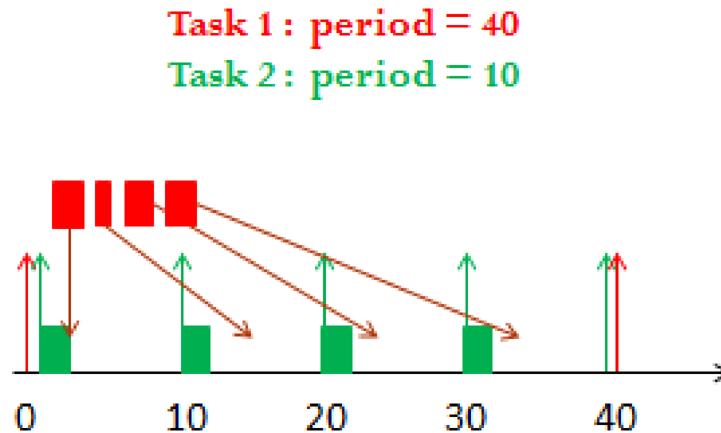


Figure 2.6: Static division of the longest task

implementation. The problem is that the execution of both tasks has to be completed in the shortest period. The task which has the longer period (usually also having the longer execution time) should be inserted during the shortest period. Hence the requirement of single-task implementation is,

worst-case execution time of task 1 + worst-case execution of task 2<shortest period

A solution for this limitation can be to statically divide the longest task 1 into smaller sub tasks and to spread their execution on several periods (as shown in Figure 2.6). The code would be then:

```
Every period = 10 do
     //Do a part of the task 1(approximatively 1/4 of the work)
     read_inputs(IN1);
     compute_outputs(IN1,OUT1);
     write_outputs(OUT1);

     read_inputs(IN2);
     compute_outputs(IN2,OUT2);
     write_outputs(OUT2);
end
```

Nonetheless this will complicate the implementation. The efficient decomposition of a task into approximatively four parts that take the same time to finish executing is difficult. Actually, timing analysis will be necessary to guarantee this requirement.

2.4 Multi-task implementation

Another solution is to switch to multi-task implementation using preemptive scheduling on single-core architectures. Tasks can be independent or communicate information with each

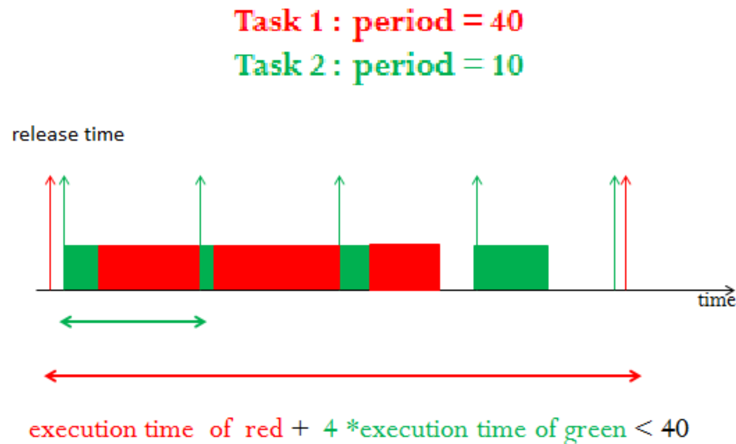


Figure 2.7: Multi-task execution

other. As we can see in the figure 2.7, the situation is improved because the longest task (Task 1) is divided into sub parts on different periods dynamically by preemption. We see that during each period of the fastest task, a part of the slowest task is executed. The latter finishes its execution before the 4th instance of the fastest task. It could also finishes its execution during the 4th instance of the fastest task. The main requirement is that the duration of execution of all the parts of the slowest task and the sum of the execution duration of the four instances of the fastest task are less than 40.

Execution time of task 1 + 4 * execution time of task 2 $<$ 40

This requirement is less strict than the requirement of the single-task implementation that obliges both tasks to be executed during the smaller period.

In conclusion, the multi-task implementation presents some advantages compared to single-task implementation. Nonetheless, more gain was demanded especially for intensive computing applications. This is why we will look at many-core architectures in the next chapter. Henceforth, we are interested by the implementation on many-core architectures where tasks are executed in parallel on different cores. However we won't put aside the implementation on single-core architectures we discussed in this chapter because, previous works on single-core architectures will inspire us for the solution on many-core architectures.

— 3 —

From Single to Many-core architectures

3.1 Single-Core architectures

The number of transistors on chips is progressing exponentially at the basis of the well-known Moore's law. Moore's law states that the number of integrated circuits doubles every approximately 2 years. It has been slightly changed to specify a period of 18 months. This law had a tremendous impact on technology since it was enunciated. The prediction seemed to be accurate and has been adopted to guide the long-term planning in semiconductor industry. It was believed as a target that should be attained in research and development sections. This exponential growth has lead to improvements in many other sides like speed and memory storage. Alternative techniques have been used benefiting from the growing number of transistors on chip. We mention techniques like: out of order execution, caching, pre-fetching, multi-threading ... The world witnessed the greatest computation capacity improvement in 1988 when it increased by 88% per year.

Unfortunately, this law cannot continue indefinitely due to limitations of the physics. Some studies expect that it will continue to be valid until 2015 or 2020 [16] when others expect earlier degradation [1]. Scaling transistor dimensions and reducing the silicon area that allow increasing the circuit frequency is limited. In fact, the continuous CPU frequency growth is not possible due to the "Hot gigahertz" phenomenon [15]. The "Hot gigahertz" phenomenon states that increasing the frequency more than it is currently implies significant heat emissions. These emissions cause a physical melt-down which is not accepted due to its catastrophic effects.

Finally, saving silicon space allowed to add more computing units which results in multi and many-core architectures.

3.2 Multi-Core architectures

A multi-core architecture contains many computing units (cores) which are all connected on chip through a bus. Cores execute codes concurrently, which offers a high-level of parallelism. Usually these replicated cores share the same memory to communicate which causes interference issues. Then, in addition to consistency protocols for memory, these architectures benefit are limited by the bus sharing. The complexity of the paradigm of shared memory generated a

new class of architectures which is the many-core design.

3.3 Many-Core architectures

Many-core architectures answer two main concepts:

- These architectures contain a high number of computing cores
- There is a way that allows these cores to communicate without bottleneck.

According to Hot Chips papers authors [19], three designs enhanced the apparition of many-core, the Cavium's Octeon fusion architecture, the SPARC64 X and the Intel's Xeon phi. The main goal of many many-core designs is to ensure a high-level of parallelism and high performance. They are presented as solutions for many intensive applications. Currently, they are used for many applications in image and audio fields, signal processing and data and networking for example.

Nevertheless, any many-core architecture is not suitable to life-critical systems. As we saw before, critical systems have very hard constraints that should be respected. Despite their powerful capabilities they are not able to fulfill these requirements. The mechanisms that has been used to enhance high performance are originally the sources of the increasing complexity of timing and functional analysis and the decreasing of its precision. It is then, becoming clear that one challenge for critical systems is the predictability. Although this criteria is not easily reachable in many-core architecture, they could not be easily replaced due to the good energy-performance tradeoff they offer. Sources of non-predictability can be cores, bandwidth resources like buses and NoCs, and storage resources like memory and caches.

This problem has been taken into account in many designs. Thus when some designs are very far from being applicable to critical applications, others seem more suitable to be used for that domain. Henceforth, we will talk more precisely about one design that has been chosen to work on, due to his characteristics in predictability domain. The chosen architecture is the Kalray MPPA-256. The problem of predictability has been taken into account when designing the architecture and implementing the Kalray MPPA-256 many-core processor.

3.3.1 The Kalray MPPA-256

The Kalray MPPA-256 is a many-core processor containing 256 computing cores and 32 managing cores. Cores are distributed among 16 computing clusters and 4 I/O clusters. Communication between the different clusters are ensured by network-on-chip (NoC). Computing clusters and I/O clusters are interconnected through two NoCs one for data (D-NoC) and the other one for control (C-NoC). D-NoC supports high bandwidth data transfers whereas C-NoC are dedicated to flow control of D-NoC. Then, NoCs established a 2D torus topology extended with direct links between I/O subsystems as shown in Figure 3.1.

Each computing cluster contains 16 cores and 1 resource manager core and each I/O cluster contains 4 cores. Inside each computing cluster (Figure 3.2), computing cores perform all

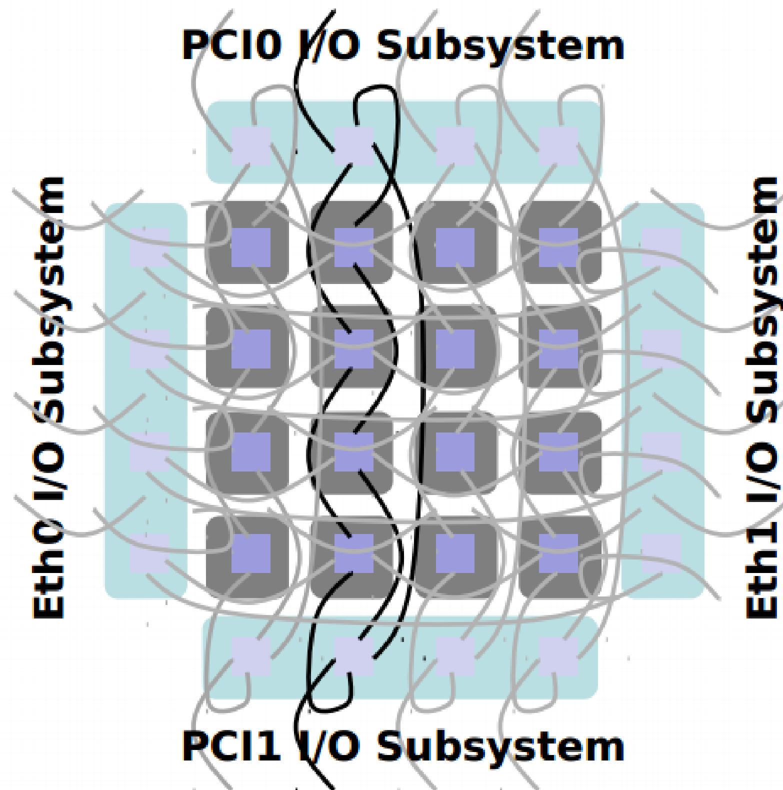


Figure 3.1: MPPA-256 processor memory spaces and dual network-on-chip (*Figure taken from the Kalray MPPA-256 manual*)

computations tasks whereas managing cores are referred to as systems core and perform resource manager tasks like scheduling tasks on the 16 other computing cores. Communication between cores of the same cluster are supported by shared memory of 2MBytes. Each core possesses its private data and instruction cache. Nonetheless, no cache coherency is imposed by the hardware.

The Kalray MPPA-256 worked mainly on the design of cores, memory system and NoCs to enhance predictability.

The Kalray's MPPA-256 Cores

The design of cores was chosen mainly to respect timing constraints and energy efficient obligation. Very long instruction word (VLIW) were used to benefit from instruction level parallelism under power and energy constraints. In addition, significant works aimed to allow to the Kalray's MPPA-256 cores to support accurate timing analysis. Accurate timing analysis allows to determine approximately how much time a piece of code will take if executed in a specific environment state. The less there are sources that may vary the duration of execution time, the more the timing analysis will be accurate. Most of the later sources have been eliminated. For example, VLIW cores don't implement out of order executions. Out of order implementation does not execute a sequence of code in the order of the sequence but decides to change the order to ensure a faster execution. However this decision is data dependent and hard to be

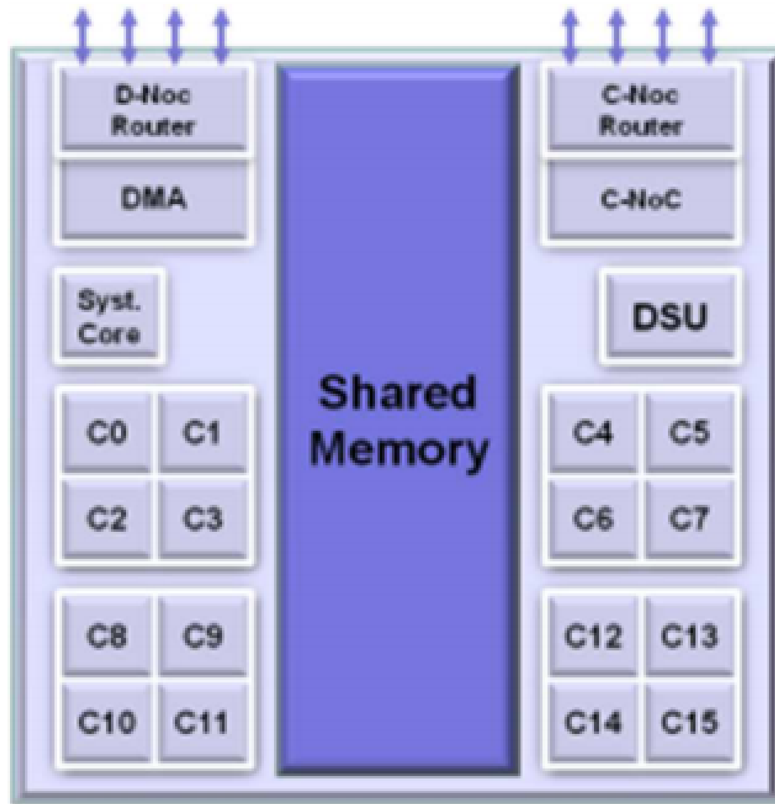


Figure 3.2: A MPPA-256 Cluster (*Figure taken from the Kalray MPPA-256 manual*)

predicted by timing analysis. In addition, other adaptations are made like choosing a suitable replacement policy for caches (LRU), suitable design for pipelines, elimination branch prediction... At this level, it is worthy to say that the VLIW cores of the Kalray MPPA-256 has been qualified as being *fully timing compositional* by the CERTAINTY FP7 project. A fully timing compositional core does not contain timing anomalies. Let's take the example of a program that contains the instruction I. The execution duration of I has an impact on the execution duration of the program. If when I takes less time to execute the program takes more time to execute then this is a timing anomaly.

The Kalray's MPPA-256 Memory system

The memory system requires also suitable design. We saw before that inside a cluster, caches are private to each core whereas memory is shared between all cores. Obviously, cores want to access concurrently the memory. Hence, if not well designed, an interference problem would appear and harm the predictability. To reduce memory problems, the Kalray's MPPA-256 memory can be configured in two ways, interleaved or blocked. The memory is partitioned in memory banks. The interleaved configuration aims to spread cores memory accesses over memory banks. The blocked configuration allows to locate data and code executed by each core on different banks. The interleaved configuration serves the high-level of parallelism whereas the blocked configuration minimizes interference between core accesses [7]. **Referring to the**

definition of the task seen above, this design makes the hypothesis of bounded execution time of a task reasonable.(ongoing work) More precisely, a task is a set of instructions and memory accesses. If memory accesses duration are not bounded, it is impossible to assume a bounded execution time task and to find its WCET.

The Kalray's MPPA-256 NoCs

Finally, we will talk about networks on chip (NoC). Traditionally bus-based architectures were used to ensure communication on chip like ARM's AMBA buses and IBM's Core connects. They have a simple topology and they are easily extensible. Nevertheless, shared buses are considered as the performance bottleneck. They have several disadvantages like non-scalability, non-predictability for wire delays and high power consumption. Also, when they are used concurrently they cause interference between users.

NoCs appeared As a consequence, networks on chip appeared. As defined in the literature a NoC is a *set of multiple point-to-point data links interconnected by switches (routers) such that messages can be relayed from any source module to any destination module over several links, by making routing decisions at the switches.* Currently, they are used to ensure communication between different clusters on the chip. They are known to offer a high-level of parallelism since links between NOCs can operate in parallel. In addition, they offer low cost area and more predictable power dissipation, speed, noise... They are also more scalable and offer better average latency than the bus architecture. Although the average latency has decreased in NoCs architectures, one transmission can witness large and indeterminable latencies between source and destination.

Problem of NoCs Although communicating via NoCs offers many advantages, the problem of transmissions witnessing indeterminable latencies remains. In this section, we take an example that illustrates this problem. Let's take the example as explained in [U.S Patent 8 619 622]. The Figure 3.3 represents one of the NoCs. It contains 5 multiplexers. Each one is associated with 4 FIFOs, one for every direction: EAST, WEST, NORTH, SOUTH. Each multiplexer directs the flow for one direction: EAST, WEST, NORTH, SOUTH. There is also an additional direction representing the local resource. Figure3.4 is a simplified network containing 8 NoCs. It shows 4 concurrent transmissions involving the NoCs. We notice that some links are more demanded than others. Link (N12-N13) is the most solicited link. It is used by the transmission (N02 - N12 - N13), the transmission (N01 - N11 - N12 - N13) and by the transmission (N10 - N11 - N12 - N13). Links (N01 - N11) and (N11 - N12) are also well demanded by existing transmissions. The Figure 3.5 shows the multiplexers N12 and N11. We notice that the FIFO FW is filling fast because it is receiving packets from N01 and N10. Unfortunately, it is not emptied because multiplexer N12 is busy emptying the FIFO containing packets from N02. Thus, the N12 router sends an overflow flow message to N11 when its FIFO containing packets from this router is full. Hence, the multiplexer N11 is no longer able to send packets. Unfortunately, transmission (N00 - N01 - N11 - N21) will then be suspended even if it is not sharing a congested link. We can easily now see that the multiplexer in N01 will overflow and prevent other transmissions like (N00 - N01 - N02).

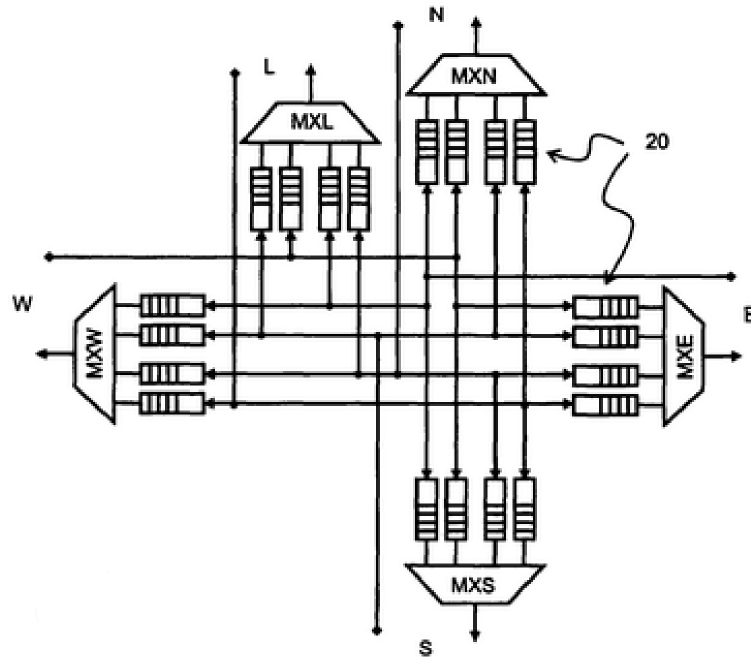


Figure 3.3: A NoC (*Figure taken from the U.S Patent 8 619 622*)

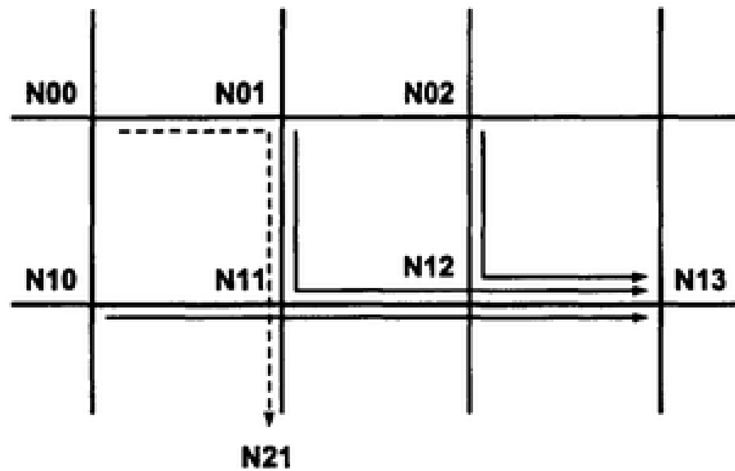


Figure 3.4: The network (*Figure taken from the U.S Patent 8 619 622*)

Problem resolved by the Kalray MPPA-256 The problem has been addressed by the Kalray's MPPA-256 design to suit real-time applications demanding quality of service and predictable communication time. The main goal is to guarantee a minimum bandwidth transmission and a maximum latency. The idea proposed by the Kalray's MPPA-256 is based on limiting the bandwidth of each transmission to prevent the described case to happen. Hence, the quota of some identified communications is such that the sum of these quotas is less or equal to the quota of the link they pass through. If the transmission capacity is limited to 18

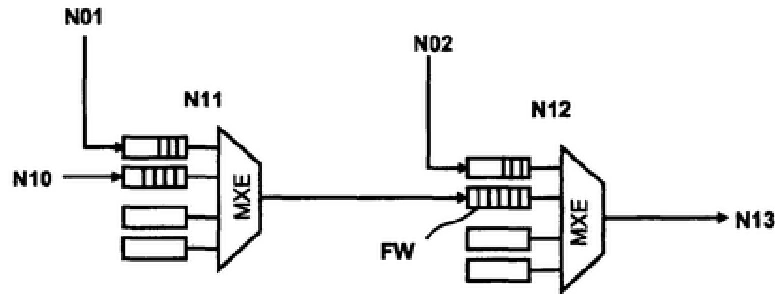


Figure 3.5: Two multiplexers involved in communications (*Figure taken from the U.S Patent 8 619 622*)

data units per time unit:

- Transmission (N02 - N12 - N13) will receive 6 units per time unit
- Transmission (N01 - N12 - N13) will receive 6 units per time unit
- Transmission(N10 - N11 - N12 - N13) will receive 6 units per time unit
- Transmission (N00 - N01 - N11 - N21) will receive 12 units per time unit

Thanks to throughput limitation, the FIFOs are never going to overflow and cause the problem described above. Hence, the flow regulation, non-blocking routers and round robin strategy at each router guarantee that each packet will be delayed by a limited time that is at most three times the maximum packet size (1 FIFO for each of the 4 directions). Thus, we can quantify how much a packet can be delayed. **As a consequence, works on the Kalray's MPPA-256 NoC allows us to assume bounded communication between clusters.**

— 4 —
Mapping high-level models (Lustre) on many-core architectures
Semantics preservation problem

4.1 The mapping problem

Usually embedded and critical systems used simple architectures like single-core architectures. This simplicity allowed to ease timing analysis and to compute the worst-case execution time (WCET). Modern critical systems which need high computation resources cannot be limited to single-core architectures. Multi-core architectures based on shared memory increase the complexity of the mission due to increasing non-predictability sources and causing tasks interference. Many-core architectures that shifted to networks on chip are the most convenient candidates, especially architectures addressing the problem of predictability, like the one built by Kalray. We start with a high-level Lustre program. The goal is to map the model on the Kalray MPPA-256. The first step is to partition the program into tasks. Several nodes can be grouped and pre-compiled into one task that has a specific period. The second step is then to decide a mapping of these tasks on the cores of the MPPA-256. Certainly, in each step many choices are made and have their impact on the implementation of the program on the Kalray MPPA-256. The third step is to ensure that the mapping of tasks on the Kalray's MPPA-256 cores guarantee determinism. This step requires adding special mechanisms to avoid non-determinism issues. The goal of this internship is, once having a specific partitioning of the high-level program and once having a specific mapping of this program on the many-core architecture, to study the impact of the implementation of the program and to find a solution for undesirable aspects.

4.2 Example

We will illustrate the goal of the internship with an example. Let's take a very simple example: a high-level program that computes the double of a flow of integers. For the input flow: 1 2 3 5 ... the output flow: 2 4 6 10 ... We suppose that the program contains many nodes interconnected. The desired implementation on the Kalray MPPA-256 should have the same semantics as the high-level program (see Figure 4.1):

For the same input flows, the high-level program and its implementation on the Kalray MPPA-256 should give the same sequence of outputs.

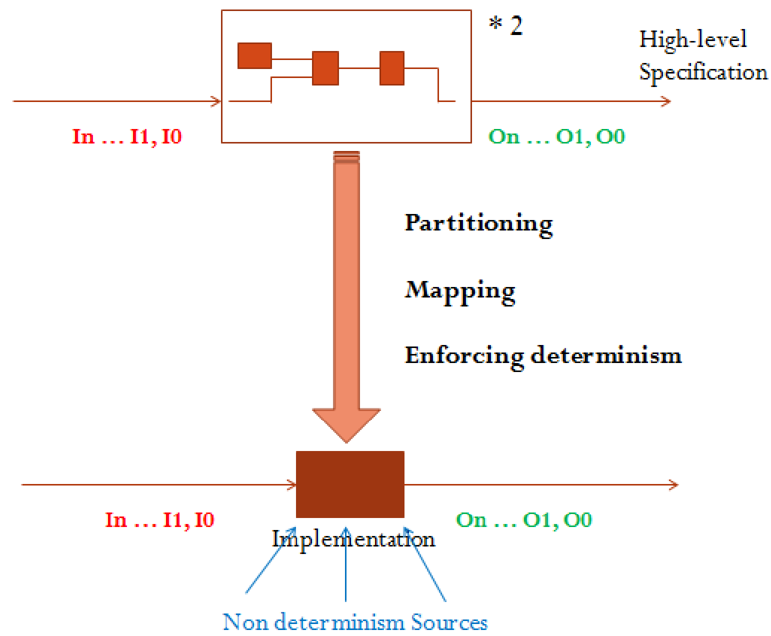


Figure 4.1: High-level specification and implementation

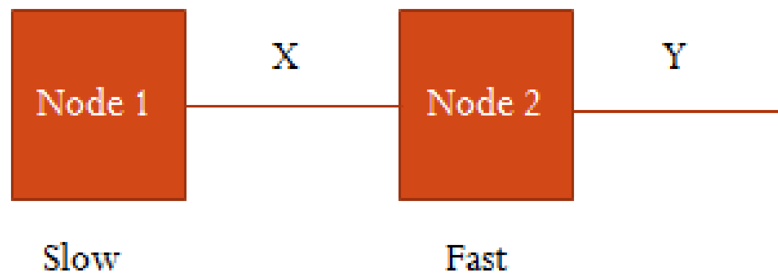


Figure 4.2: High-level program

4.2.1 Program's specification

The specification of the high-level program is: given an input flow X the computed output flow is Y = 2 * current(pre(X)). Figure 4.2 shows two computing nodes in Lustre (Each node can contain many other nodes). The node 2 compute faster than the first node. Actually the ticks of the second node occur more often than the first one. The node 1 generates the variable X. The node 2 should compute the variable $Y = 2 * X$. Nonetheless, X and Y don't have the same clock. Because X is faster than Y, there will some ticks of Y where the variable X is not defined. Operators presented in section 2.2 will be used to solve the problem. The desired specification is shown in Table 4.1. The first node is four times slower than the node 2. This is why the variable X is defined once over four times regarding the basic clock of the node 2. The **pre** operator delays the flow X. The **current** operator re-defines the value of **pre(X)** with respect to the basic clock. The node 2 always sees the same value of X four consequent times.

Table 4.1: Sampling operator

X	0	_	_	_	1	_	_	_	2	_	_	_	3	_	_	_
pre(X)	_	_	_	_	0	_	_	_	1	_	_	_	2	_	_	_
current(pre(X))	_	_	_	_	0	0	0	0	1	1	1	1	2	2	2	2

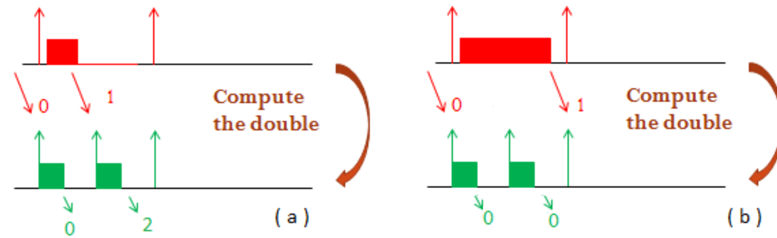


Figure 4.3: Computation duration problem

4.2.2 Elementary communications problem

As explained in chapter 2, a high-level program is implemented with a multi-task implementation. Tasks can have different or same periods. They can be independent or they can interact with each others and to communicate. The communication between two tasks involves a writer and a reader. The writer communicates his output value to the reader who uses it as input value. The difference with the implementation on a single-core architecture is that, on many-core architectures OS tasks can be mapped to different cores. The implementation faces many sources of non-determinism that could change the computed sequence of outputs and violates the original semantic of the program.

Computation duration

The first example is the computation duration. Computations are mostly data-dependent and thus their duration vary. In Figure 4.3, we show how changing computation duration can have impact on outputs. We have two tasks, the red task is a writer writing at lower rate than the green task who reads value written by the writer task. This is why the reader will read the same value many times. As we said before, we aim to have always for the same input flow the same output flow. Nonetheless, the changing duration of computation harms this goal. In the first case of the Figure (a): The input flow: 0 1 ... The output flow: 0 2 ... In the second case of the Figure(b): The input flow: 0 1 ... The output flow: 0 0 ... Then, we see that changing computation duration can have an impact on the generated sequence of outputs. In the second case the duration of computation is longer, what delays writing the value 1. Hence the second release of the reader use the older value 0.

Communication duration

In addition, the communication time can also have an impact on the computed seuence of outputs (Figure 4.4) In the first case of the Figure(a): The input flow: 0 1 ... The output flow: 0 2 ... In the second case of the Figure(b): The input flow: 0 1 ... The output flow: 0 0 ...

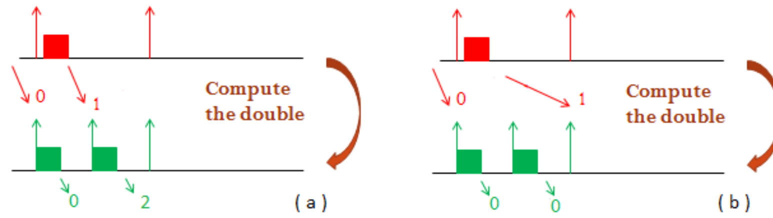


Figure 4.4: Communication duration problem

Depending on the communication duration of the written value we can be in the case (a) or case (b). If the communication duration is small the second instance of the reader receives the value 1 and outputs the value 2. Otherwise, it reads the older value 0 and outputs 0.

We just saw two parameters that for the same sequence of inputs communicated by the writer gives different sequence of outputs. The sources of non-determinism come from the architecture or the program itself.

The problem described above is the problem of **freshest value** implementation. The reader always uses the last completely written value by the writer. Having many sources of non-determinism at a given instant of the execution the last written value by the writer is not necessarily the same as for another execution. As a consequence, we want to avoid this implementation because non-deterministic implementations are not suitable for real-time applications. For instance, it is not a coincidence that the freshest value implementation cannot be expressed in a Lustre program.

— 5 —
A deterministic mechanism for the communication via the NoC

The studied problem in this report is the communication of tasks via NoCs. If many clocks are present in a program, several communication situations occur:

- Slow writer communicating with a fast reader.
- Fast writer communicating with a slow reader.
- A writer and a reader having the same period, communicating.

Figure 5.1 shows our generic example for studying NoCs. In the figure, we see one writer and three different readers. All the arrows in the Figure represent a communication via NoC between tasks mapped on different clusters. On the left side, we see tasks known as ”input provider”. Their job is to ensure the availability of data for the writer and the readers at any time they require data. On the opposite side, we see tasks known as ”output receiver”. Their job is to be ready to welcome data sent by readers at any time. This continuous availability ensures that no blocking communication will occur for communications between the input provider and writer or reader also between the readers and the output receivers. We are interested by non-blocking communication because blocking communications synchronize tasks. We see that tasks don't have the same period, then, it has no sense to synchronize all the tasks because this would mean that all the tasks will be constrained to the slowest task. For example, the reader 1 runs 4 times faster than the writer. In addition, it has another source of inputs that runs as fast as the reader. Obviously, the reader should not be synchronized to the writer especially that it has another input provider. This latter outputs data to be processed very often.

5.1 The case-study

The studied problem and the proposed solution will focus on the couple slow writer / fast reader. Figure 5.2 shows a naive implementation of the program on the Kalray MPPA-256. The writer (red task) and the reader (green task) are each one placed on different clusters. Hence, the communication is done via the NoC. The implementation we see is the freshest

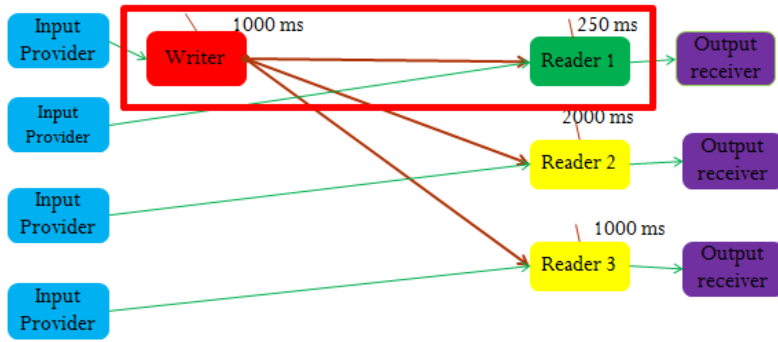


Figure 5.1: Generic example to study communication via NoCs

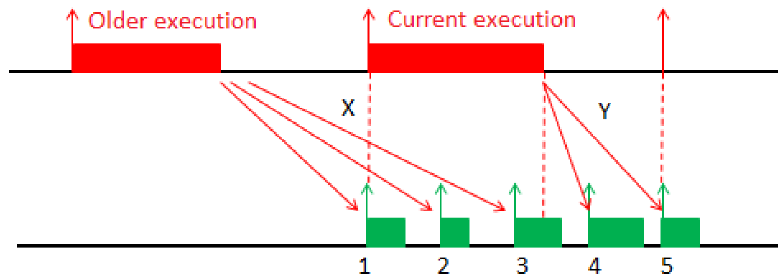


Figure 5.2: Naive implementation of slow writer / fast reader

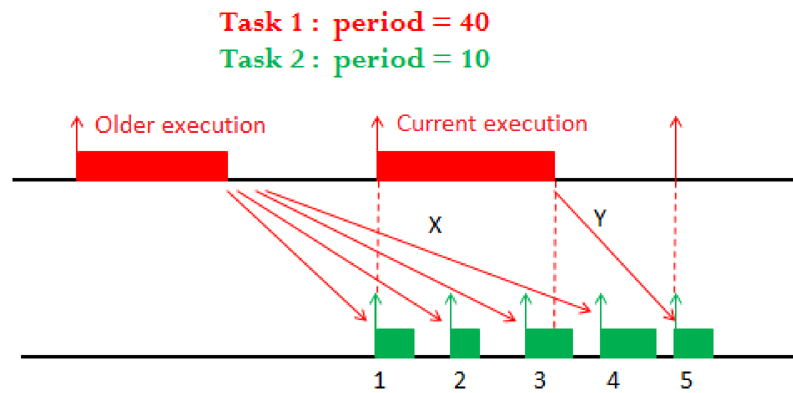


Figure 5.3: Naive implementation of slow writer / fast reader (changing communication duration)

value implementation. Instances 1,2,3 of the reader use the value X whereas instances 4,5 use the value Y. In Figure 5.3, because of a longer communication duration, instances 1,2,3,4 of the reader use the value X whereas only the instance 5 uses the value Y. Comparing the two executions the five instances of readers use different input values which obviously imply different output values in general. Hence, we cannot rely on changing NoC communication to guarantee determinism. The proposed solution is based on guaranteeing that all reader instances will use the same input values in all possible executions.

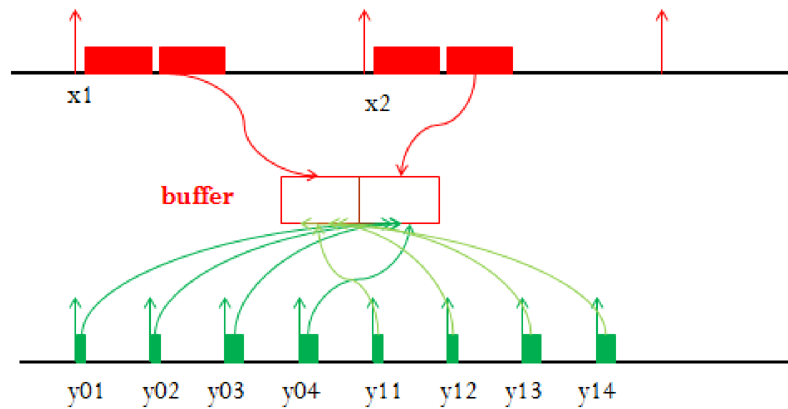


Figure 5.4: Dynamic Buffering Protocol

5.2 Previous work: Semantics-Preserving Multi-Task Implementation

The proposed solution is based on the method developed by the paper ”Semantics-preserving multi-task implementation of synchronous programs” [6]. The paper studies the implementation of a synchronous program as multiple tasks running on the same core scheduled under different strategies like fixed priority and earliest deadline first (EDF). The paper presents the inter-task communication, dynamic buffering protocol (DBP). It is a method based on intermediate buffers between communicating tasks and **write to/ read from** pointers. These pointers are manipulated on **release time** of tasks. This point distinguishes DBP from other protocols and it has been chosen because the release time is a deterministic time that does not depend on other parameters and is specified before runtime. The protocol distinguishes 3 agents:

- Writers
- Slower readers than the writer
- Faster readers than the writer

It distinguishes also two types of communication:

- Direct communication
- Communication with one unit delay

Let’s look to a simple example (see Figure 5.4): Two tasks are scheduled on a single-core architecture, the red one is the writer task and the green one is the reader task. The writer is the slowest task (longer period) and the reader is the fastest task (shorter period). Usually slow tasks are likely to have longer execution duration than fast tasks. Both tasks share a double

buffer for communication, when the writer is writing to one case of the buffer the reader is reading from the other one. At the next release time of the writer, the pointers **write to** case and **read from** case are swapped. Then, as we can see in the Figure 5.4, on the x1 execution of the writer it writes to the first case of the buffer while the reader on the y0* executions reads from the second case of the buffer. At the next release time of the writer, pointers are swapped and the writer writes to the second case while the reader in its next executions (y1* instances) reads from the first case. Therefore, the protocol allows to the reader to use the same sequence of inputs during all the executions. The basic idea to keep in mind is that the protocol relies on **releases time** to decide which value to read.

5.3 Contribution of the report

On a many-core architecture like the Kalray MPPA-256 the parallelism is real and tasks are executed concurrently on two different cores. The proposed idea is based on :

- Both tasks are aware of each other's period.
- The communication time on the NoC is bounded. The delay that a packet witnesses cannot exceed three times the maximum size of a packet.
- The execution time of a task is bounded and has a WCET.
- Tasks are harmonic: The slower task has a period that divides the faster task's period. Tasks are also released at the same time.

The idea of the double buffer presented in the paper is used in the solution. One case of the buffer is offered by the NoC connectors. The other case is a local variable maintained by the reader. The reader uses its local value until the time he is sure that the new value has been written on the NoC. When the reader reads the value on the NoC, it copies its value to his local variable. The case on the NoC is re-used by the writer to contain the next written value. This is the buffer swapping mechanism. As we can see in Figure 5.5, in the proposed solution the execution of instances y21, y22, y23 of the reader uses the value written by the instance x1 of the writer. The instance y24 of the reader was able to use the value written by the instance x2 of the writer since the computation finishes before the release of y24. Nonetheless, the solution forces the reader to use the local value instead of reading the value on the NoC because we are not sure that at this moment the value of x2 will be ready on other executions (see 5.3). Reading the value on the NoC is only done at the instance y31 of the reader because we know that for all the possible executions the value of instance x2 will be ready at this instant. This value is then copied in the local variable of the reader. The next three instances of the reader use the local value as inputs. And so on.

It is worth saying that the solution doesn't rely on preemption like the solution on the single-core. It is done with tasks running in parallel on different cores. This would ease the computation of WCET which is more complicated if the paradigm of preemption is used.

The next section will detail the implementation of the solution on the Kalray MPPA-256.

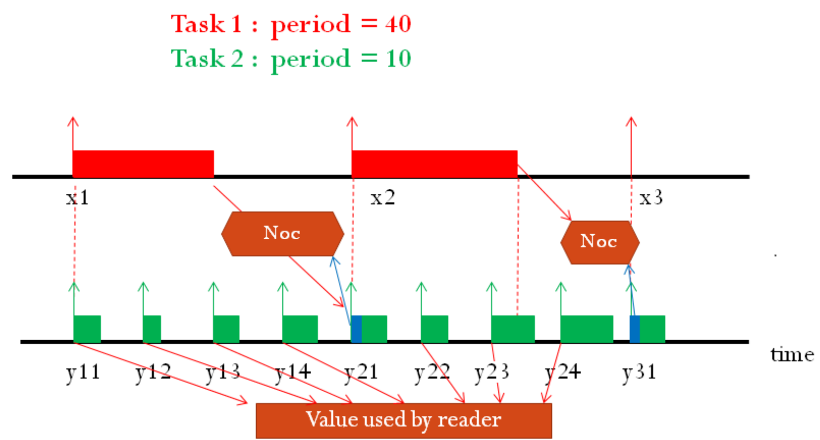


Figure 5.5: Proposed solution

— 6 —

Implementing the solution on the Kalray MPPA-256

6.1 NoC connectors for the Kalray MPPA-256

In this section, we take a look at the connectors of NoC offered by the Kalray MPPA-256 [7]. The design principles of POSIX-level (Kalray's POSIX kit) programming model are based on processes on I/O clusters that spawn sub-processes on computing clusters and pass arguments via traditional *argc* and *argv*. The communication between processes is different from other POSIX communication because it is accomplished by operating on special files, whose path-name are structured in a way that fully identifies NoCs involved in the operations of reading or writing. These connectors involve two endpoints that may have multiple transmit ports (Tx) and multiple receive ports (Rx). Five connectors are implemented. Table 6.1 summarizes the purpose of each connector, the Tx and Rx endpoints and the involved resources.

- The **Sync** connector is a half synchronization barrier. This connector involves one reader process and many writers. The reader process possesses a value **match** initialized at the beginning and blocks when it tries to do a reading operation. Each writer, at each writing operation, sends a value that is **OR**ed with the match value of the reader. When the result of the **OR** operation is -1, the reader is unblocked. If a multi-cast is needed then the connector can be also used as a barrier involving many writers and many readers. Only the C-NoC participates in this connector because few data transfers are needed.
- The **Portal** connector is a connector than involves one reader and many writers. When the reader does a reading operation, it blocks. It specifies a trigger value that indicates after how many transmissions it should wake up. Writers send data to the memory of the reader without any involvement of the later. After receiving the supposed number of transmissions the reader is unblocked. This connector can also be used in multi-cast mode and it uses D-NoC capabilities.
- The **Stream (sampler)** connector involves one or more readers and only one writer. The reader possesses a circular buffer where it receives data sent by the writer. Messages sent by the writer fill the buffer in a circular method. Older messages are overwritten even if they are not processed by the reader yet. The reader is not aware of the communication

Connector	Purpose	Tx:Rx Endpoints	Resources
Sync	Half synchronization barrier	N:1, N:M (multicast)	CNoC
Portal	Remote memory window	N:1, N:M (multicast)	DNoC
Stream	Remote circular buffer	1:1, 1:M (multicast)	DNoC
RQueue	Remote atomic enqueue	N:1	DNoC+CNoC
Channel	Zero-copy rendez-vous	1:1	DNoC+CNoC

Figure 6.1: NoC Connectors (*Figure taken from the Kalray MPPA-256 manual*)

and is not involved in the writing operation to its buffer. Nonetheless, the reader maintains a pointer to the offset of the latest received message in the buffer. This connector involves D-NoC resources.

- The **Channel** connector is a simple connector that realizes a ”rendez-vous” between a writer and a reader with zero-copy data transfer. When the reader process tries to read a value, the size of the available buffer is sent to the writer via the C-NoC. When the writer tries to write a value, it reads the size from the C-NoC mailbox. The writer computes the minimum size between the size requested by the reader and the size of the write. It sends then data via the D-NoC. After receiving the notification of end of tranfer, the reader is unblocked.

- The **Rqueue** connector involves one reader and many writers. The reader possesses a remote queue that will be supplied by the writers. A first synchronization is needed between the reader and the writers. The reader specifies a value **credit**. This value is sent to writers and allows them to write values to the reader. For example, if the writer's credit is one then writer is able to write only one time. The next time it tries to write it will block. Each time the writer writes a value the credit decreases by one. When the reader reads the data, it re-sends one credit to the writer.

6.2 Choice of a connector

To implement the communication between our two communicating tasks, we don't use the **channel** connector. Actually, the communication should not be a blocking communication. The blocking communication synchronizes the communicating tasks. Therefore, if two tasks of different periods are communicating, the fastest one will be limited by the slowest task because it will block each time the other one is not ready to communicate. Avoiding blocking using this connector is possible. For instance, we tried to add a listener task that is ready to read values written by the writer at any time. Nonetheless, adding tasks complicates the implementation. Thus other solutions using only the available tasks are prefered. The **sync** connector is also not used because of the half barrier synchronization. The **portal** connector also blocks the reader until it receives all the requested messages from the writer. The connector **sampler** is usually used to allow the reader to be able to read the latest value written by the writer. Based on its description, the connector guarantees that the reader is able to find the offset of the latest value written by the writer. This use is avoided in our implementation because it represents the **freshest value** implementation. Nonetheless, it could be used in our implementation in another way. This choice is rejected because it needs strict timing constraints.

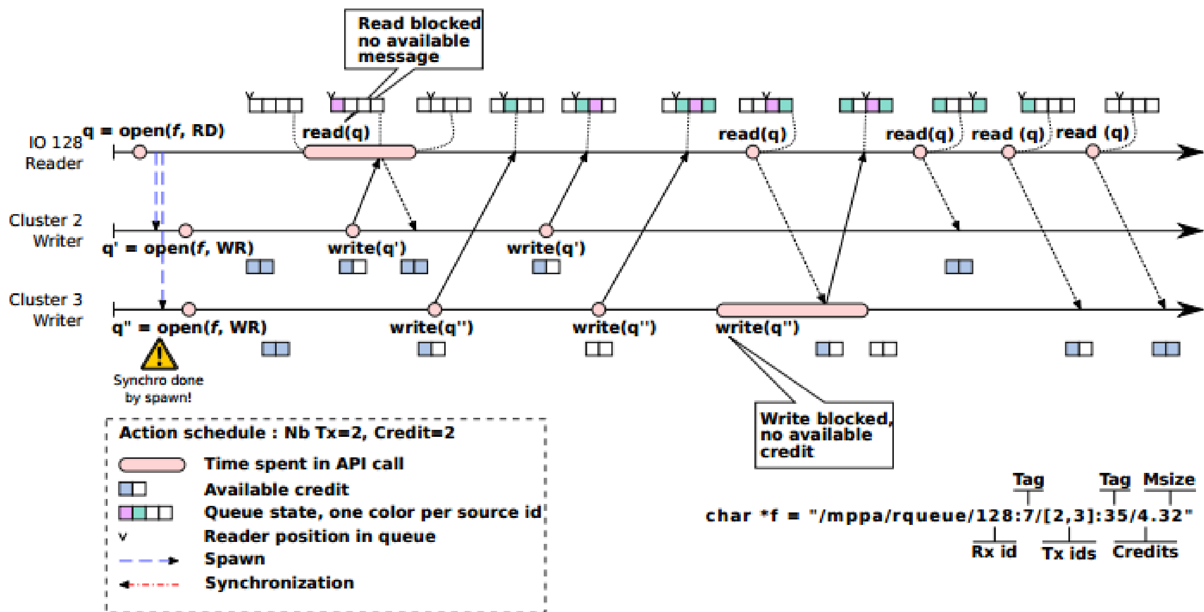
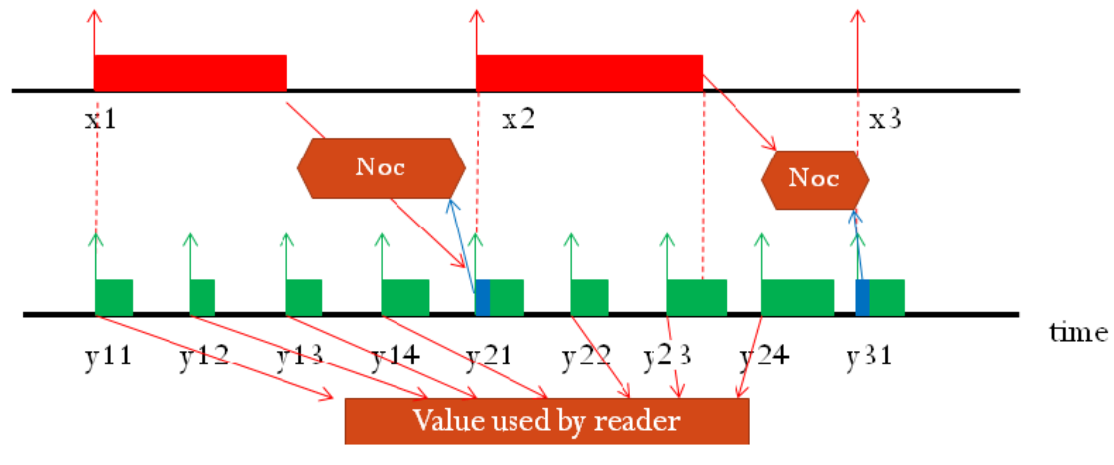


Figure 6.2: Rqueue connector (*Figure taken from the Kalray MPPA-256 manual*)

The chosen connector is the **Rqueue connector**. We illustrate the explanation about the **rqueue** connector by Figure 6.2. The Figure shows the use of the connector by two writers and one reader. The reader task is placed on the I/O cluster. The I/O cluster spawns the two writers on cluster 2 and cluster 3. This operation synchronizes the three tasks at the beginning. As we see in the pathname of the **rqueue**, the receiver side id is 128 which is the id of the I/O cluster. On the other side, the transmitters id are 2 and 3 for clusters 2 and 3. The specified credit is 4. Credits are divided equally between writers. Thus, each writer receives 2 credits. The size of the queue of the reader is also specified on the path and is 32 bytes. The queue state of the reader and the available credits of writers are also shown on the Figure. First, the reader tries to read a value from the queue. The read blocks because no available data are present in the queue. The writer on cluster 2 writes a value. The credit of the latter decreases by 1. Therefore, the reader unblocks and returns one credit to the writer who has written the value (writer on cluster 2). Many writes occur after. The writer on cluster 3 spends its two credits. Hence, when it tries to write in the third time it blocks. It is only unblocked when the reader reads one of the value it has written.

In our case, the connector involves only one reader and one writer. The writer is slower than the reader. As shown in Figure 6.3, after synchronization, the instance x1 of the writer and the instance y11 of the reader begin to execute. The reader uses the local value of the variable. To guarantee deterministic outputs for a given sequence of inputs, the reader should only read the value written by the writer when it is sure that, at the instant it reads this value, the value will be ready for all possible executions. Hence, the instances y12,y13,y14 of the reader also use the local value of the variable. If we don't apply this principle and let the instance y13, for example, read the value, a non-deterministic communication will occur as described in chapter 5. Only the instance y21 reads the value written by the instance x1 because only at this moment we are



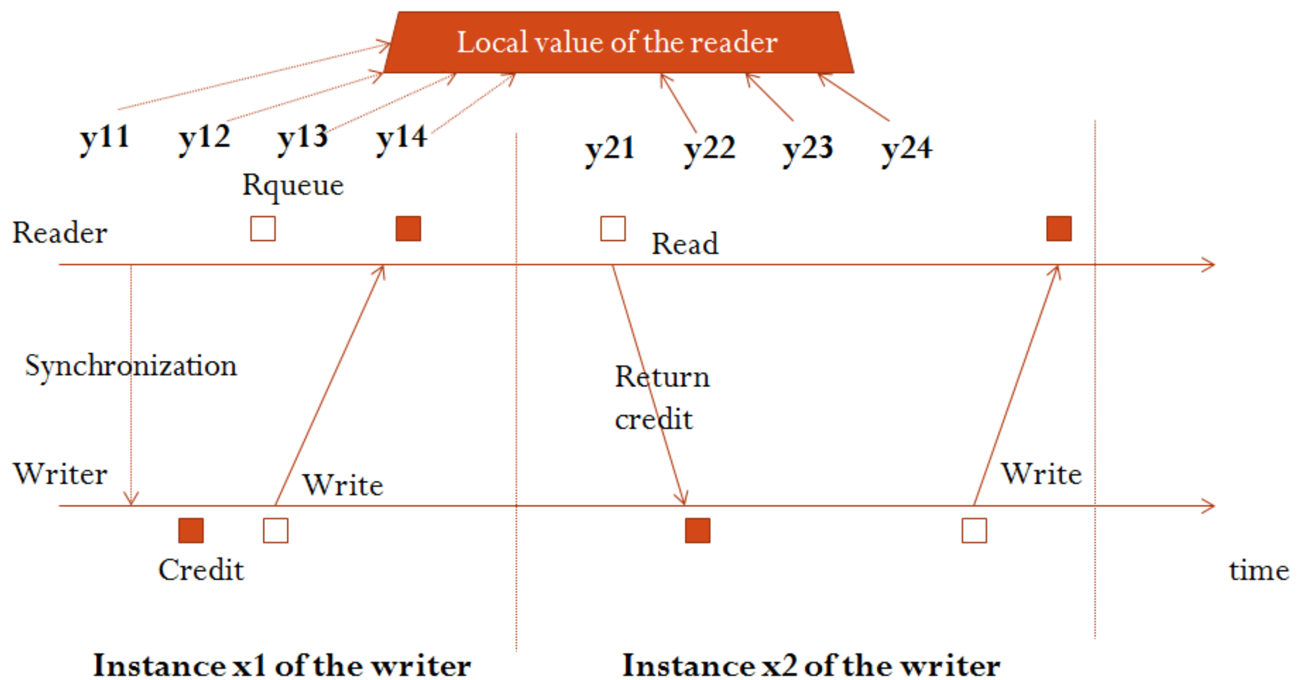


Figure 6.3: Using Rqueue connector

sure that x1 has finished writing its outputs for all possible executions. The same principle is applied for the rest of the communication. The credit is initialized by 1. The writer can then write only once without blocking. Hence, if between two writes we are sure that the value will be read at least once, then no blocking will occur. By construction of the described method, a read always occurs at the common release time between the reader and the writer which is situated between two writes. Therefore, we are sure that there will be no blocking.

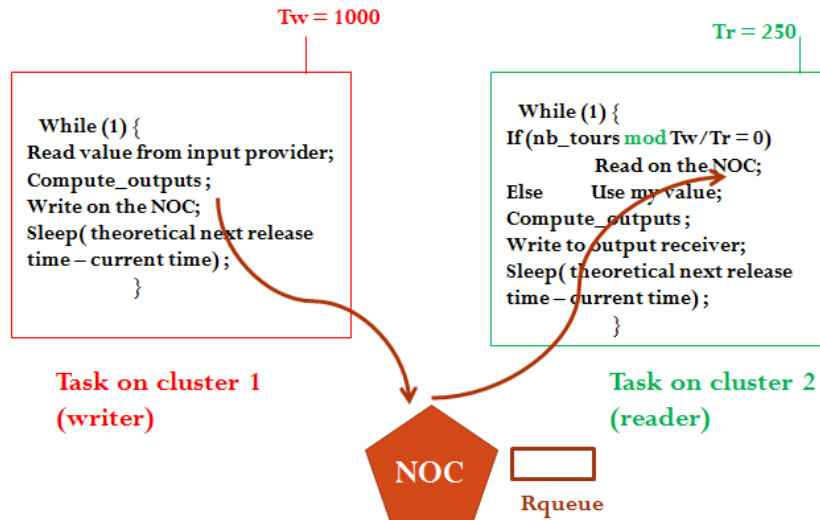


Figure 6.4: Implementation

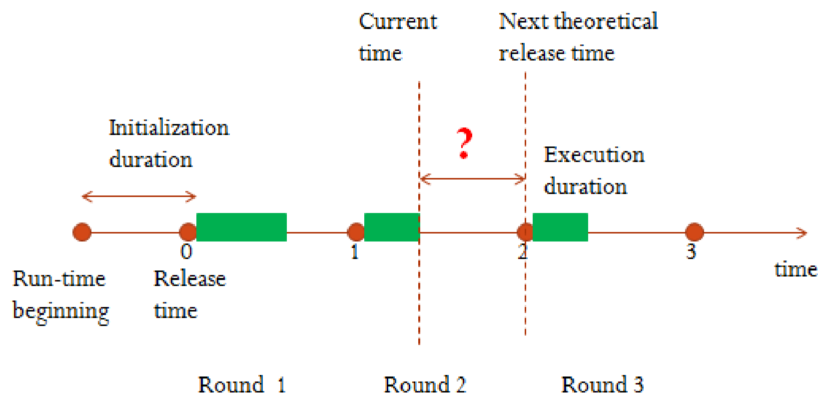


Figure 6.5: Time basis

6.3 Implementing the writer and the reader

As we see in the Figure 6.4, the two tasks are represented by two infinite loops. For each box, we associate the period of the task: $Tw = 1000$ and $Tr = 250$. The skeleton of the body loop follows the one of reactive systems which:

- Read inputs
- Compute outputs
- Write outputs

6.3.1 Time basis

The writer and reader tasks begin to execute. Both tasks begin by executing standard functions like opening the NoC connector that allows them to communicate with the desired option: **READ ONLY** for the reader and **WRITE ONLY** for the writer. They also do the appropriate operations to communicate with the **input providers** and the **outputs receivers**. They use the **Sync** connector to do the synchronization required by the use of the **Rqueue** connector. The duration of the execution of these functions is the duration of **initialization time**. Each task reads inputs, computes outputs and writes them. The duration of these operations may be smaller than the task's period. Hence, the task should sleep until the next release time. Each task maintains a variable **round** that allows it to memorize the number of round it is executing. Figure 6.5 shows the used method to compute how much a task should sleep before the next release time. For example, the second instance of the task sleeps:
duration = next theoretical release time - current time
= initialization time + period * round - current time

To get the **current time** and to **sleep** the computed duration, we use POSIX's primitives in the Kalray Kit.

6.3.2 The writer

The writer begins by reading inputs offered by the input provider. As said before, the input provider is always ready to provide inputs whenever the writer needs inputs. The writer computes the outputs and write them to the reader via the NoC. The computation must have a WCET less than period. In addition, after adding the cost of the communication to the NoC the result is assumed to be less than period. It finishes by waiting for the next release time. We use the function *usleep* to model the elapsed time between the end of the execution of the task and the next release time. The time to sleep is computed using the time basis explained above.

6.3.3 The reader

The reader follows the same paradigm. Nonetheless, it does not read on the NoC each time. If the reader is sure that the new value has been written then it reads on the NoC. Otherwise it uses its local value. As explained before, the reader task should wait until the worst case of the availability of the written value to guarantee that all the possible executions will take into account the same value. If we try to apply this idea on Figure 6.3, the reader should not read the value written by instance x2 of writer, before the instance y31 of the reader. Knowing the period of the writer, the reader should then only read on the NoC every $\frac{Tw}{Tr}$ times. In this example, the reader will then read on the NoC $\frac{1}{4}$ times as shown in Figure 6.3.

The next section presents the evaluation of the solution on the Kalray MPPA-256.

— 7 —
Evaluation

This part aims to evaluate the implementation of the solution on the Kalray MPPA-256 processor. The goal is to verify that the communication between both tasks is deterministic and for the same sequence on inputs we always obtain the same sequence of outputs.

7.1 Example

We use a simple example as shown in Figure 7.1. The writer communicates the value $2*X$ to the reader. The reader receives also another input Y. It computes and writes the output $2*X+3*Y$. X and Y are two determined integer flows. From one execution to another one these flows don't change values. The writer task is spawned on cluster 1 and the reader task is spawned on cluster 2. The input providers are spawned on clusters 3 and 4 and output receivers are spawned on I/O cluster and on cluster 5. The reader is four times faster than the writer task. Hence, it seems obvious that based on our method the reader will use the same value four consequent times. The input provider 1 gives the flow: 1,2,3 ... The input provider 2 gives the flow: 0,2,4,6 ... The first input used by the writer is 0 whereas the first input used by the reader is -1.

7.2 Instrumentation

To verify the variation of communications duration we use **time stamps**. When the writer sends the value, it attaches to the packet the time when it is sending the packet. When the reader receives the packets, it compares the time stamp of the packet with its current time. The difference between the current time of the reader and the time stamp of the packet allows us to estimate the duration of communication between the two clusters. The communication duration is longer if clusters are further from each others (in a 2D torus topology) because the packet should make more hops to reach to the cluster destination. Obviously, this method cannot be used in the proposed implementation because as seen before the reader doesn't read the value directly when it is written. Then, to estimate the communication time and observe its variation we implement another pair writer/reader. In this example, the reader reads the written value directly after being written. We reported the duration of communication for the 20th first communications in Table 7.1

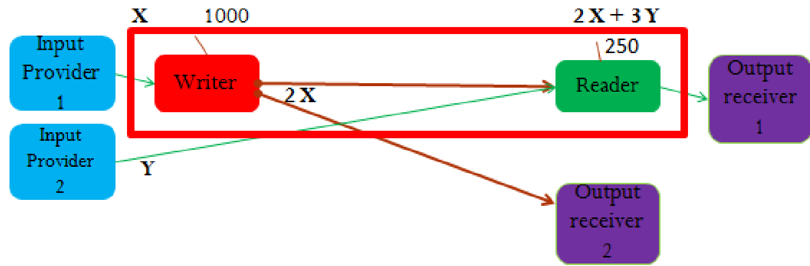


Figure 7.1: Implemented example

Table 7.1: Outputs of the proposed implementation

Duration of communication(μsec)
9040
7233
8714
8447
5276
9468
5225
7720
12456
6858
5265
6625
7134
6271
5868
8584
13818
7610
7808
7044

We see that communications duration vary with only two tasks communicating via the NoC. With more communicating tasks we expect more variability. Changing communication times is one of the reasons that justify the need of special mechanisms to enforce determinism. To comply with the hypothesis, communications time should be smaller than the smallest period. Hence, we implement tasks with period $T_w = 100000$ and $T_r = 25000$. Nonetheless, it doesn't change results regarding the studied example because the reader remains four times faster than the writer. To model a changing computation duration, we use the function **random** that chooses a random value bounded by $\frac{3}{4} * period$. We don't chose values greater than $\frac{3}{4} * period$ to respect that computation and communication duration is less than the period. The task sleep the latter chosen time during the time it is supposed to compute.

Table 7.2: Outputs of the freshest value implementation

Results	1	2	3
	-1	-1	-1
	6	6	6
	12	12	12
	18	18	18
	24	24	24
	30	30	32
	38	36	38
	44	42	44
	50	48	50
	58	56	56
	64	62	64
	70	68	70
	76	74	76
	84	80	82
	90	86	88
	96	94	96
	102	100	102
	108	106	108
	116	112	114
	122	118	120

In the scope of this internship, we validate our solution in debug mode by observing that tasks are behaving in the way we wanted them to behave. We especially use tracks and validate that tasks are reading, computing and writing outputs at the desired instants.

All the tests we did by the simulator offered by the kalray MPPA-256, has been replayed on the real platform. The results on the platform are the same as the results by simulation.

7.3 Observations

We notice the first 20 results received by the output receiver 1 with a naive implementation (freshest value). Table 7.2 summarizes the results. Therefore we see that for the same sequences on inputs (X and Y) the outputs are not deterministic. From one execution to another one, the outputs $(2X+3Y)$ are not the same.
This result is expected since as explained before the freshest value vary from one execution to another one.

Then, we add the proposed solution to enforce determinism. Table 7.3 summarizes the results. Despite the sources of non-determinism, we see that the outputs remain the same from one execution to another (in the table 3 of the tests). This layer forces the reader to read values at determined instants of the execution which result in the same read sequence by the reader from one execution to another one. We tried also to change the mapping of tasks by changing the

Table 7.3: Outputs of the proposed implementation

Results	1	2	3
	-1	-1	-1
	5	5	5
	11	11	11
	17	17	17
	24	24	24
	30	30	30
	36	36	36
	42	42	42
	50	50	50
	56	56	56
	62	62	62
	68	68	68
	76	76	76
	82	82	82
	88	88	88
	94	94	94
	102	102	102
	108	108	108
	114	114	114
	120	120	120

clusters they are spawned on. This help us to increase communications duration if communicating clusters are not neighbors. The results remain always the same (second test). In the third test, we run the test on the real platform. No changing result is observed. The results are the expected results: The initialized value read by the reader from the writer is -1. This value will be used 4 times. In these 4 instances of the reader, the values read from the input provider are 0, 2, 4, 6. Hence, the computed values by the reader (computing $2*X+3*Y$) are:

$-1+3*0=-1$
$-1+3*2=5$
$-1+3*4=11$
$-1+3*6=17$
The second value written by the writer is $2*X=2*0=0$ that will also be used by the reader for four times. Thus, the computed values by the reader (computing $2*X+3*Y$) are:

$0+3*8=24$
$0+3*10=30$
$0+3*12=36$
$0+3*14=42$

7.4 Conclusion

Even if all the tests succeeded, this doesn't guarantee and doesn't proof the correctness of the solution. The method should be validated by formal verification. Nonetheless, we are not yet ready for such step. To do so, the hardware should first be modeled if we wanted to validate the solution by model checking as for DBP in the paper [6].

— 8 —
State of the art

The apparition of many-core architectures had founded for many works. Some works aim for critical systems and others for non-critical systems.

8.1 Non-critical systems

Obviously non-critical systems don't have hard constraints like critical systems. Usually, in non-critical systems the most searched criteria is performance. Some works targeted the automatic mapping on multi and many core architecture to achieve high performance. In [12], Servet Benchmarks are used to explore the most performing mapping. Servet is a set of benchmarks that explores the architecture and determines the most significant parameters influencing the performance of the mapping. Detecting the most relevant parameters for a specific architecture, the mapping of the code tries to avoid the bottlenecks of the architecture. Most important evaluated parameters are cache hierarchies, bandwidths, memory accesses and communication between cores. Results show that the algorithm used has highly increased the efficiency and the performance of the mapping without penalizing the application and without requiring any code changes. Other automatic mapping algorithms like AMTHA are designed to answer the need of minimizing the execution time of an application. As described in [8], the algorithm leverages the model of parallel application MPAHA and the abstraction of the underlying architecture to find a suitable mapping that guarantee good execution times. The algorithm is also capable of estimating the results of the mapping. Results shows that the algorithm is able to estimate execution times with 6% error.

The most studied aspect for designing these algorithms on architectures like multi and many core architecture is resource sharing between mapped tasks. Shared resources are classified into two categories the bandwidth resources like NoCS ans buses and storage resources like memory and caches [17]. Because of the performance is a main concern on these systems, the impact of resource sharing on performance and on performance prediction has been studied in many works. In this context, we mention the murphy [5] approach that for every access to a shared resource assumes the worst cost it can include. This approach is known to be pessimistic and give very high upper bound on timing analysis. Another alternative is to use the slowdown factor approach. The approach consists in estimating the performance of the application in an isolated system. Then, measurement-based approaches are used to estimate the slowdown fac-

tor. This factor will be added to the isolated analysis to give an estimation to the performance of application while co-existing with other applications. Measurement-based approaches leverage stressing benchmarks who are a set of benchmarks related to one particular resource. These benchmarks aim to quantify the maximal slow down that a task can witness co-existing with other tasks using the same resource. These methods contribute to estimate the execution time bounds for many and multi core architectures where the resource are highly shared between cores. It helps to improve mapping analysis and to find more optimized mappings.

8.2 Critical Systems

In the other side, non-critical systems cannot accept performance at the cost of less deterministic systems. Some works addressed the problem of shared resources by avoiding the interferences. To avoid the interferences, they re-designed new deterministic hardware. Between these works, we mention the Composable and Predictable Multi-Processor System on chips (CoMPSoc) [14]. The main motivation behind this idea is the exponential increase in the complexity of functional and temporal verification with the increasing number of applications on chip. Hence, the idea is based on removing interference through ”resource reservation”. Thus, every application can be analyzed independently of other applications. In the same domain, we mention the case of precision timed machine (PRET) [10].The authors argue the shift to PRET because architectures are fighting for average-case performance ignoring the predictability and repeatability of timing analysis. The most harming consequences were seen in embedded computing. The extreme example that was repeated in Avionic is the example of fly-by-wire that translates the commands of the pilot and transmit them to the actuators. It's difficult for this software to be validated. In addition, any slight change requires all the software to be revalidated due to lack to composability. The goal of PRET is then to build a suitable architecture for high-reliability embedded applications. Although the main requirement is determinism, performance is not forgone. PRET leverages some existing techniques like extending the instruction set by instructions delivering accurate timings, using scratchpad memories instead of caches, adding timing semantics to programming languages...

Our work is then placed between both sides; between the first campaign who uses many-core architectures and benefit from high performance without caring about predictability and the second campaign who wants to re-design the hardware and generate a new deterministic hardware. The techniques of the first campaign are not applicable due to the strict determinism required by critical systems. In addition, the solution of the second campaign is expensive. The intermediate solution is to not use imperatively a full deterministic hardware but a hardware that addressed the problem of determinism in their design like the Kalray MPPA-256 did in designing cores, memory system and communication between cores (NoCs). Determinism is then retrieved by using some methods like the one proposed in the report.

Other works have studied the mapping of data-flow models on multi and many-core architectures like synchronous data-flow models, cyclo static data-flow models and schedulable parametric data-flow. Data-flow models are mainly used for multi and many-core architectures because of their ability to express the high-level of parallelism of these architectures. Hence, We mention works for heterogeneous multi processor systems on chip in [18]. They introduced the MAPS tool (MPSoC Application programming studio). The goal of this work is to develop

a solution for optimized temporal and spatial task to processor mapping. In addition, works in [20] aimed to develop a library to express communication between actors in data flow graphs because it is a very important element in mapping data-flow models on many-core architectures. Similarly, recent works in [2] have developed a compilation toolchain to map extended cyclo-static data-flow program on many core architectures. Some works have studied their static analysis. Different problems of data-flow languages have been discussed like liveness (parts of the systems will compute indefinitely), bounded memory (the system can be implemented with finite memory), maximizing throughput and minimizing buffer sizes for a given throughput. These problems were discussed for cyclo-static dataflow in [3] and [4]. Liveness and bounded memory were addressed in [11] for parametric data-flow MoC. Efficient algorithms were derived for checking them. The goal of these works is to guarantee these properties statically to be efficiently compiled on multi and many core architectures.

Our work is not placed at the same level with other works mapping data flow programs on many-core architectures. As seen above, we describe applications as a set of tasks with different paces communicating with each others. This description allows us to have more expressiveness. In particular, it allows us to specify how often a task should compute outputs. For example, in an aircraft there are tasks that should be computed more often than others. Hence, Tasks related to the reactor are urgent tasks that should not be postponed.

— 9 —
Perspectives

Mastering the mapping of a high-level program on a many core architecture is early. Table 9.1 positions our work in the set of remaining works. We are working on communications via NoCs. Three cases should be studied:

- One pair of tasks of different periods (Here we are studying slow writer to fast reader)
- One pair of tasks of same period
- Different pairs communicating. Obvioulsy the problem of interference is to study for this case.

The cases are to study between clusters (communication via NoC) and inside the same cluster (communication via shared memory). The problem of tasks scheduled on the same core of the same cluster is already widely discussed in previous works. An application is partitioned in tasks that will be spread on the Kalray MPPA-256, on same or different clusters, on same or different cores. After studying all these cases and evaluating the predictability of each proposed solution, we could also study their performance. Even though our main focus is predictability, poor performing solutions are not accepted in modern systems. The impact of each mapping on application performance is then to study. For example, mapping two communicating tasks on two neighbor clusters is more performing than mapping them on further clusters. Once the impact of each mapping is studied, the results will help in the decision of tasks mapping on the architecture. An upper problem should then be studied which is to study how to partition a high-level program into tasks. If we work on Lustre for instance, it should be decided what nodes can be grouped together and be pre-compiled in one task. The work should be also extended to support non-harmonic tasks. Hence, we see that it is a complex and a challenging problem that will stay around for a long while. These works can also be extended to be supported by other architectures.

Figure 9.1: Perspectives

— 10 —
Summary and Conclusions

The current hardware architectures, designed to answer average performance needs, are not suitable for critical systems. Actually, mechanisms used to boost high-performance are best-effort mechanisms that don't take into account the predictability and determinism issues. Others, like the Kalray MPPA-256, have addressed these issues in their design to fulfill certification requirements. Nonetheless, some problems remain and might prevent users from reaching the desired level of determinism. Among these problems, we mention the communications on chip via the networks on chip (NoC) that we choose particularly to study because it is the less studied element in previous works.

In this report, we presented a deterministic mechanism for the communication via the NoC on the Kalray platform. By observation, we saw how our proposed method have succeeded in achieving a deterministic communication between two tasks (slow writer fast reader).

Although this solution resolves only an example of a communication that may occur between two tasks, it affirms that building determinism on non-fully deterministic hardware is possible. Even if we don't know if there will be a satisfactory and a complete solution in the near future, this subject seems promising. Thus, it is worth trying to look at this problem at least to estimate how far we will be from an ideal solution. Moreover, we should not forget that acceptable solutions should also take into account performance criteria. Will future solutions be able to deal with determinism and performance at the same time ?

Bibliography

[1] Semiconductor Industry Association. *National technology roadmap for semiconductors*. SIA, 1997.

[2] Pascal Aubry, Pierre-Edouard Beaucamps, Frédéric Blanc, Bruno Bodin, Sergiu Carpov, Löıc Cudennec, Vincent David, Philippe Dore, Paul Dubrulle, Benôıt Dupont de Dinechin, et al. Extended cyclostatic dataflow program compilation and execution for an integrated manycore processor. *Procedia Computer Science*, 18:1624–1633, 2013.

[3] Mohamed Benazouz, Alix Munier-Kordon, Thomas Hujsa, and Bruno Bodin. Liveness evaluation of a cyclo-static dataflow graph. In *Proceedings of the 50th Annual Design Automation Conference*, page 3. ACM, 2013.

[4] Bruno Bodin, Alix Munier-Kordon, Benoît Dupont De Dinechin, et al. Periodic schedules for cyclo-static dataflow. In *Proceedings of the 11th IEEE Symposium on Embedded Systems For Real-time Multimedia*, pages 105–114, 2013.

[5] Frédéric Boniol, Hugues Cassé, Eric Noulard, and Claire Pagetti. Deterministic execution model on cots hardware. In *Architecture of Computing Systems–ARCS 2012*, pages 98–110. Springer, 2012.

[6] Paul Caspi, Norman Scaife, Christos Sofronis, and Stavros Tripakis. Semantics-preserving multitask implementation of synchronous programs. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(2):15, 2008.

[7] Benoît Dupont de Dinechin, Duco van Amstel, Marc Poulhies, 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.

[8] Laura De Giusti, Franco Chichizola, Marcelo Naiouf, Armando De Giusti, and Emilio Luque. Automatic mapping tasks to cores-evaluating amtha algorithm in multicore architectures. *International Journal of Computer Science Issues (IJCSI)*, 7(4), 2010.

[9] Francois-Xavier Dormoy. Scade 6 - a model based solution for safety critical software development. In *Proceedings of the 4th European Congress on Embedded Real Time Software*, pages 1–9, 2008.

[10] Stephen Edwards and Edward A. Lee. The case for the precision timed (pret) machine. Technical Report UCB/EECS-2006-149, EECS Department, University of California, Berkeley, Nov 2006.

[11] Pascal Fradet, Alain Girault, and Peter Poplavko. Spdf: A schedulable parametric dataflow moc. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2012*, pages 769–774. IEEE, 2012.

[12] Jorge González-Domínguez, Guillermo L Taboada, Basilio B Fraguela, María J Martín, and Juan Tourino. Automatic mapping of parallel applications on multicore architectures using the servet benchmark suite. *Computers & Electrical Engineering*, 38(2):258–269, 2012.

[13] Nicolas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous data flow programming language lustre. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.

[14] Andreas Hansson, Kees Goossens, Marco Bekooij, and Jos Huisken. Compsoc: A template for composable and predictable multi-processor system on chips. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 14(1):2, 2009.

[15] Intel. Hot gigahertz, February 2014.

[16] Michael Kanellos. New life for moore law. *CNET News. com*, 2005.

[17] Jan Reineke and Reinhard Wilhelm. Impact of resource sharing on performance and performance prediction. In *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014*, pages 1–2. IEEE, 2014.

[18] Weihua Sheng, Jeronimo Castrillon, Anastasia Stulova, Maximilian Odendahl, Rainer Leupers, and Gerd Ascheid. Programming heterogeneous mpsocs using maps.

[19] Ron Wilson. From multi core to many core: Architectures and lessons, 2012.

[20] Mingkun Yang, Suleyman Savas, Zain Ul-Abdin, and Tomas Nordstrom. A communication library for mapping dataflow applicationson manycore architectures. In *Proceedings of the 6th Swedish Multicore Computing Workshop*, 2013.