Master of Science in Informatics at Grenoble
Master Mathématiques Informatique - spécialité Informatique
option Parallel Distributed Embedded Systems

Towards Semantics-Preserving Implementation of Synchronous Programs on Many-Core Architectures

Amaury Graillat

24th of June 2015

Research project performed at Verimag

Under the supervision of:
Florence Maraninchi and Matthieu Moy (Verimag)

Defended before a jury composed of:
Alain Girault
Noël Depalma
Martin Heusse
Jean-Claude Fernandez
Arnaud Legrand
Olivier Gruber

June 2015

Remerciements

En premier lieux, je remercie vivement Florence Maraninchi et Matthieu Moy pour leur écoute, leur aide, ainsi que pour la relecture de ce rapport. Je remercie Pascal Raymond pour sa précieuse connaissance de Lustre et ses points de vue théoriques toujours très intéressants.

Je remercie l'équipe pédagogique du master MoSIG ainsi que l'équipe pédagogique de l'Ensimag pour leurs cours instructifs et motivant.

Je tiens aussi à remercier Yohan, les doctorants ainsi que les stagiaires de la plateforme pour leurs conseils et nos discussions.

Abstract

Reactive systems are systems that constantly react to their environment. One must guarantee that the computation time of the system is bounded and matches the input-output latency imposed by the environment (for instance the period of measure of the altitude for a flight control system). We talk about multi-periodic systems when the nodes of a program are running at different periods. Today, reactive programs are more and more complex, and we reach the limit of the single-processors in term of computation. Consequently, the size of the program is limited to match the time requirements. Most of the processors are multi or many-core. But they are complex and the Worst-Case Execution Time of a program executed on these architectures is hard to compute without excessive pessimism. Yet, the Kalray MPPA many-core architecture is predictable and allows to bound both the execution time and the transmission time between the many-core clusters.

Our purpose is to run synchronous programs on many-core architectures with bounded delays. On one hand, we consider a synchronous program composed of several nodes communicating together, on the other hand a many-core chip composed of several clusters and a Network-On-Chip (NoC). The first problem is to find a way to map the nodes on the clusters. The other problem is to give the semantics of the communication between the nodes. The semantics must be deterministic and preservable when implemented on the many-core architecture to guarantee equivalence between the synchronous program and the implementation. We give a deterministic semantics to communications between the nodes of different periods. We give the implementation of a multi-periodic program on the Kalray MPPA. Finally, we evaluate our solution by testing. Our method provides a key element toward the deterministic implementation of reactive programs on many-core architecture: a way to preserve a centralized deterministic semantics of a multi-periodic program, when the program parts exchange information via the NoC.

1 **Introduction** 4
1.1 Reactive Systems 4
1.2 Many-core Architecture 5
1.3 Implementation of Synchronous Program on Many-core Architecture 6

2 **Reactive Systems and Synchronous Programming Languages** 7
2.1 Reactive Systems 7
2.2 Programming Reactive Systems 7
2.3 Lustre 8
2.3.1 Variables and Flows 8
2.3.2 Nodes 9
2.3.3 Clocks 9
2.3.4 Example Reactive Program 9
2.3.5 Multiperiodic Systems and Communication-by-sampling 10
2.4 From Simple-loop to Real-Time OS Implementations 11
2.4.1 Simple-loop Implementation 11
2.4.2 Improved Static-scheduling 12
2.4.3 Dynamic Scheduling 13

3 **The Kalray Multi-Purpose Processor Array Architecture (MPPA)** 15
3.1 Overview 15
3.2 Network-on-Chip 16
3.3 Memory 18
3.4 Individual Cores 19
3.5 Using the Kalray's MPPA for Reactive Systems 19

4 **Problem statement** 21
4.1 Running Synchronous Programs on Many-core Architectures 21
4.1.1 How to Map the Nodes on a Many-core Architecture? 21
4.1.2 How to Define a Deterministic Semantics of Data Exchanges Between the Nodes? 21
4.1.3 Previous work 23
4.2 Problem Definition 23
4.2.1 Defining a Deterministic Semantics of Communication 23
4.2.2 Preserving the Semantics of the High Level Description in the Many-core Implementation 25

5 The unit-delay semantics: deterministic and preservable data-exchanges between nodes 27
5.1 Definition of a Task . . . 27
5.2 Communication Pattern . . . 27
5.3 Computing Communication Patterns . . . 29
5.3.1 Initialization of the Reader . . . 30
5.4 Verifying the Availability of Values . . . 30
5.4.1 Simple case $p_w > WCET_w + WCTxT$ . . . 31
5.4.2 General case $p_w > WCET_w$ . . . 32
5.5 Communication Buffer . . . 32
5.6 Buffer Size . . . 32

6 Implementation of a multi-periodic synchronous program on the MPPA 35
6.1 General Principles of Implementation . . . 35
6.2 Case Study . . . 35
6.3 Low Level Communications on NoC . . . 36
6.3.1 Reader side . . . 36
6.3.2 Writer side . . . 37
6.4 Patterns . . . 38
6.5 Implementation of the Non-Drifting Period . . . 38
6.6 Spawning of the Nodes on the Clusters . . . 38
6.7 Programs Synchronization . . . 39

7 Evaluation 40
7.1 Reference . . . 40
7.2 Variation on Sources of Nondeterminism . . . 40
7.2.1 Variation of Computation Time . . . 41
7.2.2 Variation of Transmission Time . . . 41
7.3 Performance Evaluation of the Low Level Communications . . . 41
7.4 Results and Conclusion . . . 42

8 Related Work 44
8.1 Specification Languages . . . 44
8.1.1 Prelude . . . 44
8.1.2 ForeC: a Synchronous and Parallel Language . . . 45
8.1.3 Cyclo-Static Data Flow . . . 45
8.2 Synchronous Programs on Multi- or Many-core Architectures . . . 46
8.2.1 Simulation of Distributed Synchronous Programs in Shared Memory . 46
8.2.2 Specific Hardware Improved for Off-line Mapping . . . 46

9 Conclusion and future work **47**

A Appendix: Reference **48**

B Appendix: Implementation **50**

— 1 —

Introduction

1.1 Reactive Systems

Reactive systems are systems that continuously react to their environment at a speed determined by this environment. Control systems such as flight control systems or heater control system are examples of reactive systems. The timing constraints of the reactive systems are determined by the environment. Meeting them is critical for some applications (for instance the flight control system). For example, in the heater control (presented in Fig. 1.1), we do not sample the temperature with the same speed as the emergency stop button. The system must stop if someone presses the button even briefly, hence the sampling rate must be able to catch this short even. But, the temperature of a room is a physical value which has slow changes. We must ensure that the reactive program is able to execute sufficiently fast to tolerate these timing constraints.

Today, the reactive programs are more and more complex in terms of computational power. We reached the limit of the single-core processors. Furthermore, most of the current processors are multi- or many-core. Running the reactive programs on this kind of processor would give more computation power.

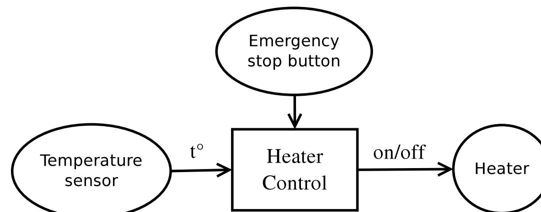


Figure 1.1: Educational example of a reactive system controlling a heater. It controls the heater in function of temperature thresholds. The emergency button stops the heater.

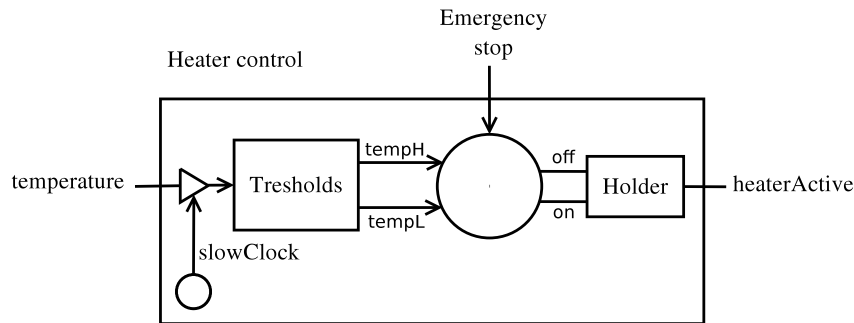


Figure 1.2: This figure represents an (educational) heater control system. It is a node with two inputs: the Emergency stop signal and the temperature value. The output is a signal to enable the heater. The circles represent the computations of the node, the triangle represents the sampling operator `when`, and the rectangles are sub-nodes.

Reactive programs are often composed of several nodes communicating together. From the timing constraints of the system one can assign execution periods to each part of a synchronous program. For instance, the node comparing the temperature with the thresholds (cf. Fig. 1.2) can run every ten seconds or so, whereas the period of the part managing the emergency button must be of 10 ms or so. To be able to execute a task at each period of time, we must ensure this task completes during this period. The computation of the Worst-Case Execution Time (WCET) is essential in the timing constraint verification.

A multi-periodic reactive system is composed of several nodes running at different periods. One can wonder what happens if a node generates values on a different rhythm than the node consuming these values. One way to give meaning to the communication is the communication-by-sampling principle. The basic implementation which consists in making a node communicating as soon as the computation finishes, and another node taking the last available value is called "freshest value" and is not deterministic because it depends on the tasks execution time (cf. Section 2.4.3).

The work of Caspi *et al* [3] explains how to fit more complex synchronous programs on one processor. The solution is based on a RTOS and the program can be multi-periodic. It defines a deterministic protocol based on buffers to implement communication-by-sampling. But, even with this solution, we reach the hardware limit of the single core in terms of computational power.

1.2 Many-core Architecture

Most of the recent processors are multi- or many-core. But, most of them are complex and optimised for average performances and thus make the WCET computation very pessimistic, if at all bounded.

A many-core architecture able to handle real-time programs is a possible successor to single processors because it offers more computational power. The many-core architectures bring several advantages compared to the multi-core architectures because the cores are more isolated. Hence, they avoid the cache coherency problem happening when several cores of a multi-core communicate through the shared memory. Furthermore, in many-core architectures, each core is simple (hence often more predictable).

The Kalray MPPA architecture is the only many-core processor on the market guaranteeing both bounded execution time and bounded communication time. Each core has neither branch prediction nor complex pipeline, and the caches have the Least Recently Used (LRU) policy which is known to be predictable. The access to the shared memory can be bounded since there is no cache coherency mechanism and the shared memory is divided into independent banks to remove interferences. A Network-On-Chip allows communication between the clusters with bounded transmission time thanks to a flow control mechanism guaranteeing latency and throughput.

1.3 Implementation of Synchronous Program on Many-core Architecture

The purpose of this work is to implement synchronous programs on such a many-core architecture. It raises several problems:

1. How to split and map a synchronous program on a many-core architecture?
2. How to define the semantics of data-exchanges semantics between the nodes (communication-by-sampling)?

The first problem is out of the scope of this thesis. We consider that the program is split into several nodes and the placement of these nodes on the clusters of the many-core architecture is given.

Our work is focused on the second problem. Namely, we consider two nodes A and B executing on different periods. If A generates values and B consumes values. As the periods of the two nodes differ, we need to define the relation between the consumed and the produced values. This relation must be deterministic (a counter-example is the “freshest value”).

Contribution Let us consider a set of nodes of a synchronous program, the execution periods and the wires between the nodes. Furthermore, we consider given a mapping on the many-core architecture.

- We give deterministic semantics for the data-exchanges between nodes of a multi-periodic synchronous programs.
- We provide a method to implement this semantics on a many-core architecture and to verify that the semantics holds when implemented on the architecture.
- We provide an implementation of a multi-period synchronous program on the Kalray MPPA.

Section 2 presents the reactive systems. Section 3 shows why the Kalray MPPA architecture is a good choice to implement reactive programs. Section 4 defines the problem of implementing a reactive program on the Kalray MPPA. Section 5 presents the solution. Section 6 gives the details of implementation. And Section 7 evaluates the correctness of our solution. Section 8 gives the related work and Section 9 gives some conclusions and future works.

— 2 —

Reactive Systems and Synchronous Programming Languages

2.1 Reactive Systems

Reactive systems are systems that continuously react to their environment at a speed determined by this environment. For instance, a control system is a reactive system. To illustrate our explanation we present an example of a heating control system.

Fig. 1.1 gives an overview of this system. The purpose is to turn the heater on and off. The first input is from the temperature sensor, the second is an emergency stop button. The heater is controlled according to an hysteresis: it is turned on when the temperature is below a low threshold and turned off when the temperature is above a high threshold. In any case, if the emergency button is pushed, the heater stops.

The environment influences the sampling rate required for the two inputs. For this system, sampling the temperature with a period of some seconds is sufficient, however the emergency stop button can be pressed briefly, so a faster rate is needed to avoid losing the event.

2.2 Programming Reactive Systems

A reactive system can be programmed with an infinite loop (Fig. 2.1 gives an example code). We remark that each operation (read, compute and write) of a reactive program takes time to execute. The duration of one iteration of the loop defines the sampling rate of the inputs, it must be sufficient for the sampling rate imposed by the environment.

As soon as the system becomes complex, programming directly with an infinite loop becomes tedious and error-prone. Hence, specific languages such as Lustre have been designed.

```
init mem;
while(true) {
    inputs = read_sensors();
    outputs = compute(inputs, mem);
    write(outputs);
    update(mem);
}
```

Figure 2.1: Example code for a reactive program. It loops forever and as fast as possible. Another implementation is to transform the loop into a periodic task (as in Fig. 2.2).

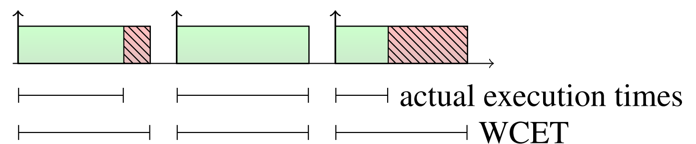


Figure 2.2: The vertical arrows represent the activations of the while loop body. The Worst-Case Execution Time (WCET) is the upper bound of the execution time. We represent three possible execution times.

```
node compute(x, y:
    int)
     returns (o: int);
var
    t: int;
let
     o = t * y;
     t = (x - 1);
tel
```

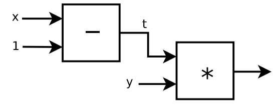


Figure 2.3: The semantics of a Lustre program is close to that of a logical circuit. The code defines equations between inputs and outputs. Equations can be derived from the dataflow diagram naming each wire.

2.3 Lustre

Lustre [10] is a dataflow language designed to program reactive systems. The program can be seen as an equational system between inputs and outputs. As shown in Fig. 2.3, a Lustre program is analog to a logical circuit. As for a logical circuit, instantaneous cycles are forbidden (*e.g.* `o = 2*o` is forbidden).

2.3.1 Variables and Flows

In Lustre, a variable is a flow of values indexed by natural numbers. This flow represents successive instants in time.

Operator pre The `pre` operator gives the previous value of an expression. An example is given below. This operator allows to express a variable as a function of its previous values in the flow, but instantaneous cycles are forbidden. Example: `X = Y * pre(X)` is allowed, but `X = Y * X` is forbidden.

Logical instants of X	0	1	2	3	4	5
X	x_0	x_1	x_2	x_3	x_4	x_5
pre(X)		x_0	x_1	x_2	x_3	x_4

In the first instant, `pre(x)` is not defined.

Operator "−>" This is an initialization operator to define the first value of a flow. See example:

X	x_0	x_1	x_2	x_3	x_4	x_5
Y	y_0	y_1	y_2	y_3	y_4	y_5
pre(X)		x_0	x_1	x_2	x_3	x_4
Y−>pre(X)	y_0	x_0	x_1	x_2	x_3	x_4

```
node holder(on, off: bool) returns (o: bool)
let
    o = false -> (pre(o) or on) and not off;
tel
```

Figure 2.4: Example of node showing how a value can be computed with a previous value in the flow.

2.3.2 Nodes

In Lustre, the code is structured into nodes. The code Fig. 2.4 shows an example node.

A node can be called in an expression. For instance `X = holder(b1, b2)` calls the node `holder`. If a node is called several times, several instances of it are produced (each with its own memory).

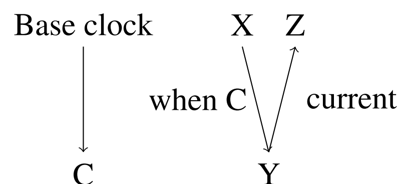
2.3.3 Clocks

In Lustre, a clock is a flow of Boolean values. Every flow in Lustre is defined on a clock. There exists a clock tree and a base clock which is the root of the tree.

Operator when The `when` operator allows to sample an expression at the rhythm of a clock. In the table below, `X` is a flow. The flow `Y` takes the values of the flow `X` when `C` is true and is not defined otherwise.

Operator current For a flow `Y` defined on a sub-clock `C` of the root clock, this operator adds the missing instants in the resulting flow (`Z = current Y`) to match with the clock of the expression As clocks are organized in a tree, if the clock of an expression is not defined, it takes the base clock of the node. For the missing instants of the source flow, the command holds the last value.

Logical instants of X	0	1	2	3	4	5
X	x_0	x_1	x_2	x_3	x_4	x_5
C	1	1	0	1	0	1
Y = X when C	x_0	x_1		x_3		x_5
Logical instants of Y	0	1		2		3
Z = current Y	x_0	x_1	x_1	x_3	x_3	x_5
Logical instants of Z	0	1	2	3	4	5



2.3.4 Example Reactive Program

We study an example reactive program. We consider the heater system described in Section 2.1, and the associated code in Fig. 2.5.

`Heater control` is a node with two sub-nodes `Thresholds` and `Holder` (cf. 1.2). The sampling rate of the inputs `emergency` and `temperature` is not the same. We choose to sample the temperature 10 times slower than the emergency button. First, we create the flow `slowClock`, true one period of the base clock out of ten. This flow is a sub-clock of the base clock since its rythm is on the base clock.

```
node thresholds(temp: int)
    returns (tempH, tempL: bool)
let
    tempH = (temp > 25);
    tempL = (temp < 15);
tel

-- executed on fastClock
node system(emergency: bool; temperature: int)
    returns (command: bool)
var
    cnt: int;
    slowClock: bool;
    (tmpH, tmpL: bool) when slowClock;
    heaterOn, heaterOff, stop: bool;
let
        cnt = 0 -> (pre(cnt) + 1) mod 10;
        slowClock = (cnt = 0);
        (tmpH, tmpL) = thresholds(temperature when slowClock);
        stop = holder(emergency, false);
        heaterOn = current(tmpL) and not stop;
        heaterOff = current(tmpH) or stop;
        command = holder(heaterOn, heaterOff);
tel
```

Figure 2.5: The control part of the example heating system. `TempH` indicates whether the temperature is above the high threshold and `tempL` when the temperature is below the low threshold. The temperature is sampled at a slow rate ($\frac{1}{10}$ of the fast rate), whereas the emergency signal is sampled at a faster rate. `Command` triggers the heater. An emergency signal can stop the heater at any time, once triggered, this signal stops the heater forever. Code of `holder` is in Fig. 2.4.

In Lustre, the clock of a node is defined by the clock of its inputs. The input of the node `thresholds` is `temperature when slowClock`, hence, this node is enabled on the clock `slowClock` and the outputs `tempH` and `tempL` are also on `slowClock`. `Stop`, `emergency` and `command` are on the base clock.

Now we give an example of how `current` converts the rhythm of an expression from a sub-clock to the base clock. The expressions to compute `heaterOn` and `heaterOff` are on the base clock whereas `tmpL` and `tmpH` are on `slowClock`. Hence, we need the operator `current` to adapt `tmpL` and `tmpH` on the base clock.

2.3.5 Multiperiodic Systems and Communication-by-sampling

We introduce the notion of multiperiodic systems. In the heater control system example (cf. Fig. 1.1), the emergency button must be sampled fast enough (*e.g.* less than 100 ms of period) to catch the signal of the button pressed briefly, whereas the temperature can be sampled with a period of several seconds. We can choose to split the program into several parts running at different speeds (we call them tasks). This kind of system is called *multiperiodic* [2].

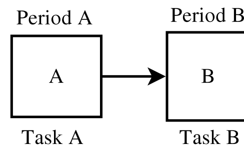


Figure 2.6: Generic example of a multiperiodic system composed of a node A and a node B working at different speeds (Period A, and Period B). We implement each node with a task.

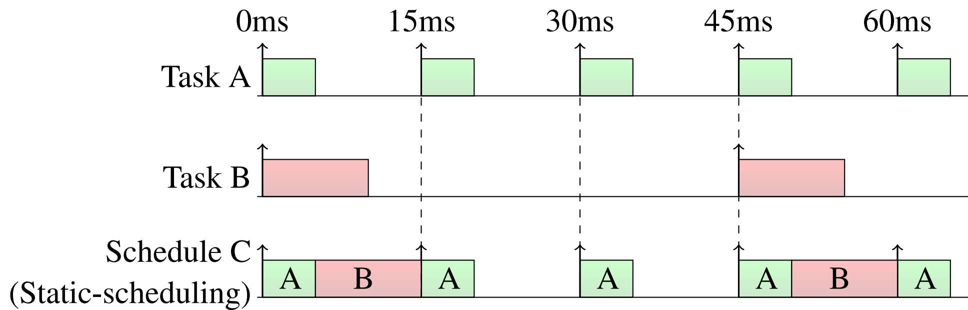


Figure 2.7: The vertical arrows represent the activations of the two tasks. Task A has a period p_A = 15 ms, Task B has a period p_B = 45 ms. The rectangles are the respective WCETs of the tasks. The lines Task A and Task B show how the tasks could execute in full parallelism. Schedule C is a static schedule of the two tasks on one processor. We execute the whole reactive program at the fastest rate (period of A). B is executed every 3 periods of A. It implies strong timing constraints: $WCET_A + WCET_B < p_A$.

Considering a generic example depicted by Fig. 2.6. B is reading values from A. If task A is faster than B, it produces values more often than B can read. Communication-by-sampling is a choice for the semantics of the communication: we decide that values can be lost. B “samples” the output of A.

2.4 From Simple-loop to Real-Time OS Implementations

2.4.1 Simple-loop Implementation

The purpose of synchronous languages is to ease the development. There exists a Lustre compiler able to generate simple loop code [10] as in Fig. 2.1.

Fig. 2.2, shows the execution of a reactive program. The execution time of a synchronous program grows with the complexity of the program. The inputs are read at the beginning of the activation, and written at the end. Verifying whether a reactive program satisfies the time specification (for instance the input-output latency) is done by computation of the Worst-Case Execution Time (WCET) of the body of the loop.

The first attempt to implement a multiperiodic reactive program is to schedule its tasks statically to execute the whole program at the speed of the fastest rate (*e.g.* sampling the temperature at the same speed as the button). Fig. 2.7 gives an example where we execute the task A at every activation of a fast clock and a task B every three activations of A. But, there is a strong timing limitation since the WCET of the two tasks must fit into the fast period and it hides some form of communication-by-sampling.

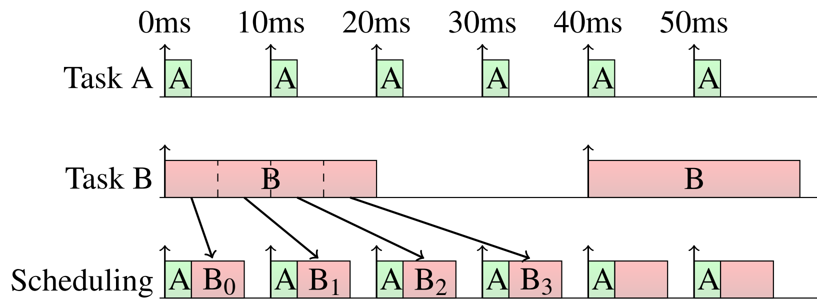


Figure 2.8: The vertical arrows represent the activations of the two tasks. The rectangles are their respective WCETs. This figure shows an example of static scheduling. The lines Task A and Task B show how the tasks could execute in full parallelism. Line Scheduling is an improved static schedule of the two tasks on one processor. Task B is split into 4 parts. At each activation, we execute A and a part of B. The constraint is that Task B must complete before its next activation.

How to use this property to fit larger systems on one processor?

2.4.2 Improved Static-scheduling

An improved scheduling could be to split the slowest task into several chunks. Imagine you could split B into B_0, B_1, B_2, B_3 of approximately equal execution times. A solution with two tasks appears in Fig. 2.8. Task A has a period of 2, and Task B has a period of 8. The WCET of A is 0.5 whereas the WCET of B is 4. Task B is split into 4 sub-parts of WCET 1. The schedule has a WCET of 1.5. It is composed of an activation of A and a part of B. For the implementation, we can use a counter to know which part of B to execute at each activation of AB.

This solution makes the execution of multiperiodic program possible even though the entire program cannot be executed at the fastest rate. Yet, it leads to several problems. The first is, splitting a task into several parts with a precise WCET can be difficult. Second, if a period or a program changes, we need to reconsider the splitting.

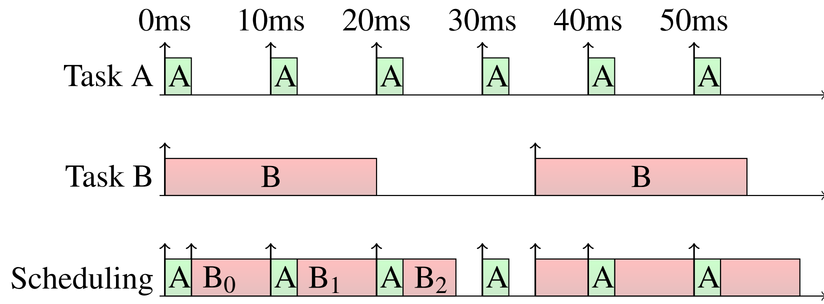


Figure 2.9: Dynamic scheduling of the tasks A and B. A high priority is given to the fastest task (Task A).

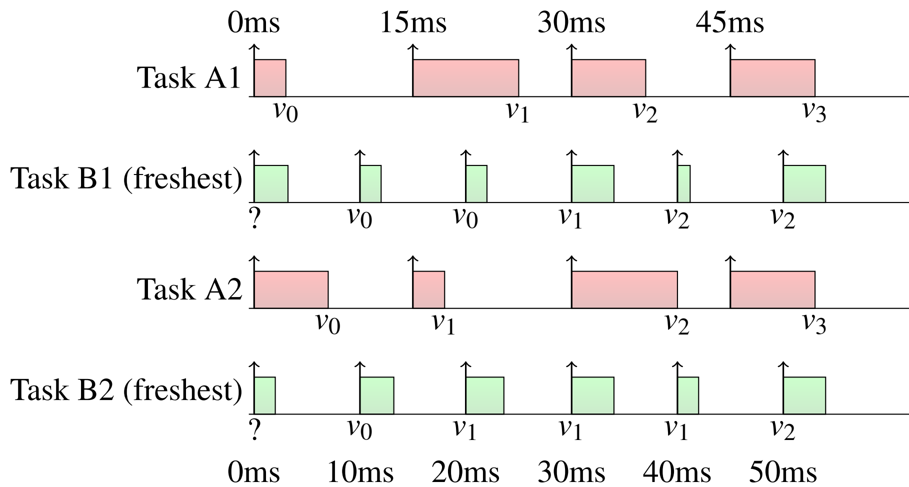


Figure 2.10: For each task, we represent the actual execution time of the tasks ($\leq$ WCET). The tasks read values at the activation and write at the end of computation. For two different executions of Task A and Task B, the sequence of value read by Task B is not the same. In the first case Task B1 reads the values v_0,v_0,v_1,v_2,v_2 from A1, whereas in the second, Task B2 reads v_0,v_1,v_1,v_1,v_2. This shows that the "freshest value" semantics depends on the computation time, and thus is not deterministic.

2.4.3 Dynamic Scheduling

We can use a scheduler (and hence a Real-Time OS) with preemption to activate the tasks. If we give a greater priority to high-rate tasks and a low priority to low-rate task, we obtain a result similar to the static-scheduling except the splitting is done dynamically (cf. Fig. 2.9), according to the actual execution times. This solution has been formalized by Caspi *et al* [3].

Yet, having several dynamically-scheduled tasks raises an important question. We know, the semantics of the data-exchanges between the nodes are defined by the synchronous program. Furthermore, the semantics of a synchronous program is deterministic (the outputs depends only on the inputs). How to preserve the semantics with a dynamic scheduler?

The first data-exchange policy we can think of is the "freshest value". It is an attempt to implement the communication-by-sampling semantics. But this policy does not guarantee the determinism since the values depend on the actual execution time of the tasks [3]. This issue is presented in Fig. 2.10.

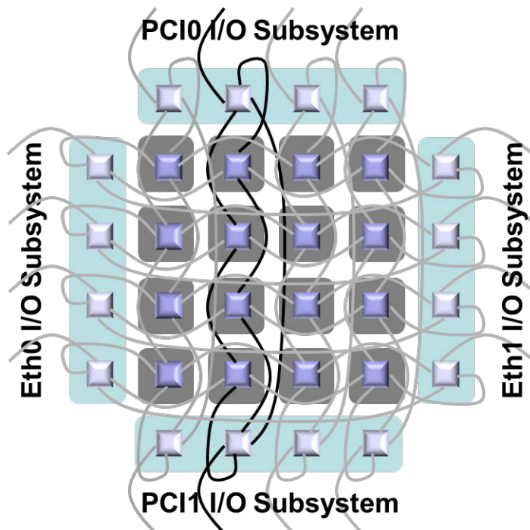
Caspi *et al.* [3] present the Dynamic Buffering Protocol (DBP). The purpose is to preserve the original semantics of the synchronous program even though the tasks are dynamically scheduled. The solution is based on shared buffers. The result is, for any execution time and any scheduler, the implementation always preserve the behavior of the simple-loop implementation of the synchronous program.

— 3 —

The Kalray Multi-Purpose Processor Array Architecture (MPPA)

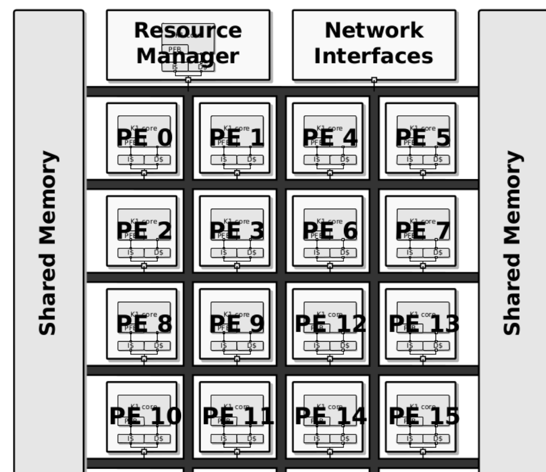
3.1 Overview

The Kalray Multi-Purpose Processor Array 256 (MPPA-256) Andey is a low power processor intended for high performance [6]. As shown in Fig. 3.1, it is composed of 16 clusters and 4 Input/Output (I/O) clusters. The clusters are connected through two networks-on-chip (NoC) with a 2D torus topology (cf. 3.3): the Data-NoC (D-NoC) is designed to maximise throughput, and the Control-NoC (C-NoC) is designed to minimize latency. Section 3.2 gives more details about the NoC.



(Figure from [6])

Figure 3.1: Overview of the MPPA-256's architecture. 16 clusters and four I/O clusters. The Ethernet (Eth) interfaces allows communication through the network. The MPPA can be used as an accelerator card connected to a host with the PCIs bus.



(Figure from [6])

Figure 3.2: Overview of a cluster of the MPPA-256. A cluster is composed of 16 Processing Elements (PE) and one Resource Manager. Each shared memory is shared between 16 PEs. The Network Interface connects the cluster to the NoC.

Each cluster is composed of 16 processing elements (PEs) plus one resource manager (RM) and a shared memory of 2 Mbytes (cf. Fig. 3.1). The RM is designed for NoC communications.

As shown in Fig. 3.1, the Kalray's MPPA-256 is equipped of two Ethernet controllers for network applications and two PCI buses to configure the processor or to use it as an accelerator card.

The cores have a VLIW (Very Long Instruction Word) architecture allowing instruction-level parallelism. Hence, the MPPA provides three levels of parallelism: the instruction (ISA) level, the PE level (communicating through shared memory), and the cluster level (communicating through networks on chip).

A typical usage of the MPPA is to load a piece of program into an I/O-Cluster. This program spawns and launches processes on the clusters (often in the Resource Manager of the clusters), then these processes can create threads on each PE.

The inter-thread communication is done through the shared memory, and the inter-process communication through the NoC. The computed data can be forwarded to the outside (host processor, network), using the PCI buses of the I/O clusters or the Ethernet cards.

3.2 Network-on-Chip

Each cluster owns a router which is connected to four other routers (North, South, West, East) as depicted in Fig. 3.3. A local link connects the cluster to its router. There are two independent networks, the Data-NoC and the Control-NoC, with the same topology.

To go from one cluster to another, the data passes through two or more routers. When two clusters are sending data, if the routes share at least one router, there is a risk of congestion. Each router has one output buffer per direction. The packets are split into 32-bits flits (unit of transmission). A router can transmit a flit in one cycle on a link. Hence, if we do not control the flow, the buffer could become full and could lose packets.

Flow control The NoC has been designed to be fair and to guarantee latency and throughput [7]. The hardware offers a budget-based flow control mechanism. The principle is to limit the bandwidth at the source (in the DMA of the clusters, see Fig 3.3). The flow control guarantees the latency and throughput of the communication with a correct configuration. Hence, if we set all the limits to their maximum, the service is not guaranteed.

In each cluster, we must configure a budget b and a window w. The DMA sends a packet only if during the last w cycles of transmission, it has sent less than b flits. Consequently, a cluster cannot send data when the budget is spent. The values w and b must be configured before the first transmission. If the values are changed during the communication, the behavior is undefined.

Fig. 3.4 shows this mechanism for two clusters. Both cluster have a budget of 10 flits during a window of 20 cycles such as for any time window of 20 cycles, each cluster does not send more than 10 flits.

The flow control mechanism is essential to ensure guaranteed service and to bound the transmission time. These properties are not available in all many-core architectures. For example, the NoC of the Nostrum is "best effort" [7] *i.e.* the packets are send as soon as possible, without control.

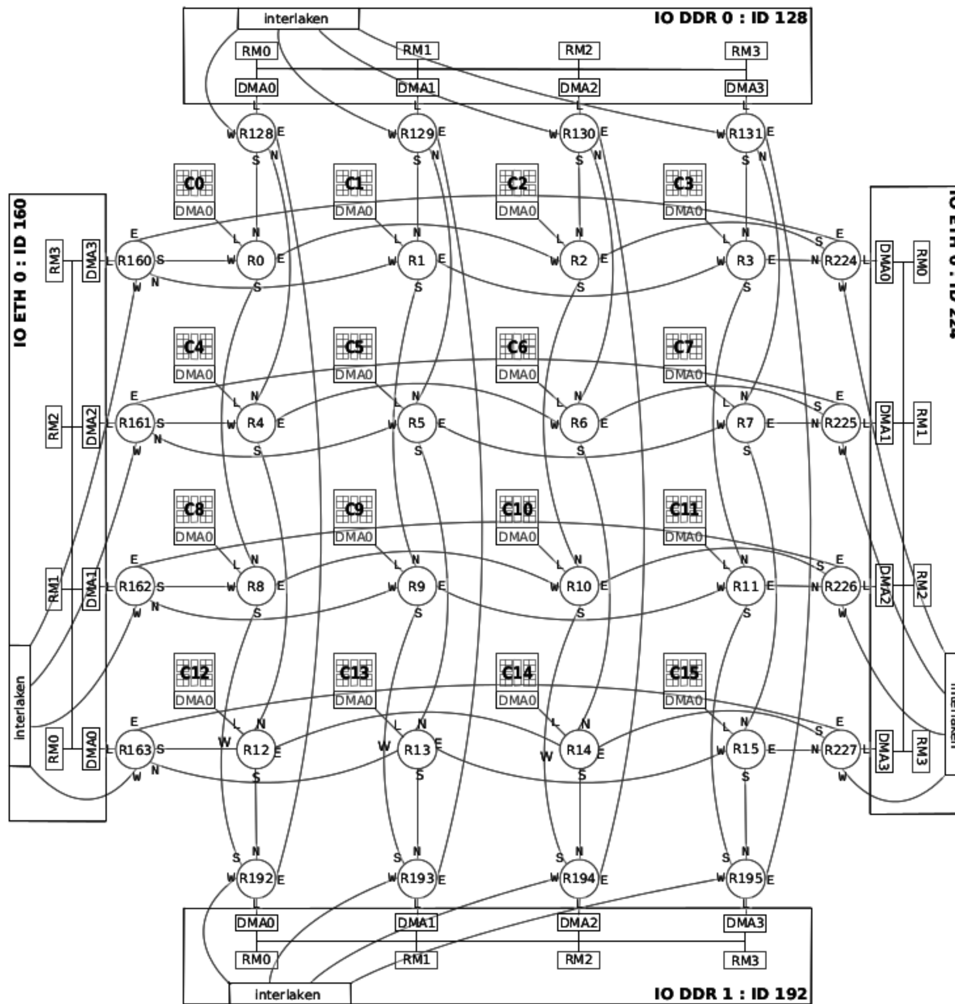


Figure 3.3: The MPPA-256's NoC topology (figure from [11]). Each cluster (C0 to C15), and each RM (RM0 to RM3) of the I/O Clusters owns a router and a Direct Memory Access (DMA) to write the packets in the memory. It has two full-duplex links with each neighbor (North, South, East, West): one for data, one for control.

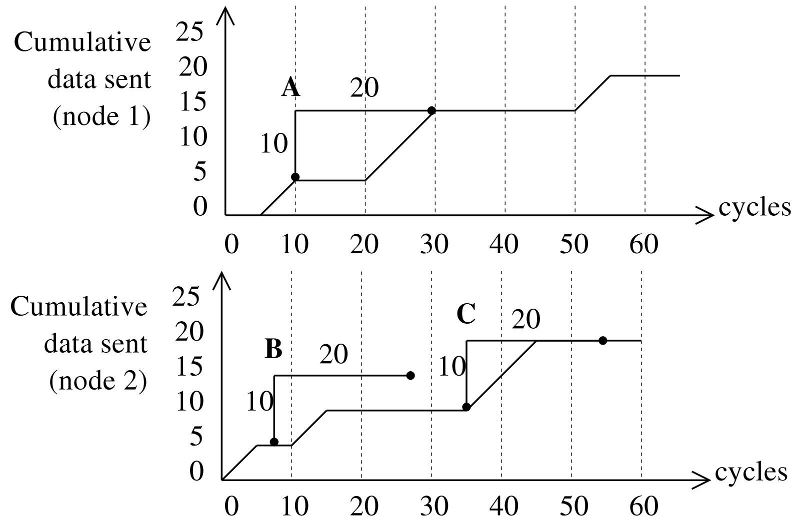


Figure 3.4: Example of flow control mechanism configured on two nodes. The x-axis is the number of cycles, the y-axis is the cumulative number of 32-bits flits sent. The two nodes have a budget $b = 10$ cycles in a sliding window $w = 20$ flits. Hence, they cannot send more than 10 flits in a sliding window of 20. For Node 1, A represents the window from t=10 to t=30. At t=30, Node 1 emission is blocked (it already sent 10 flits in the last 20 cycles). Node 2 is blocked before the end of the window C. The window B, shows that Node 2 does not use all the budget: it could send 5 flits more.

Worm-hole buffering policy The policy of the NoC is worm-hole. It means a router does not store the packets. It decides the appropriate direction, for retransmission or delivery, once it receives the header. If the outgoing link is free the transmission can start even though the packet has not been fully received yet.

The MPPA provides a communication mechanism based on the definition of a route. A route is a list of directions (North, South, East, West) corresponding to the path between a source cluster and a destination. When a packet enters the router of a cluster, the header of the packet indicates the direction to take next.

Example of route from C2 to C11: East, North, South (cf. Fig. 3.3). This route passes through the routers R2, R224, R3 and R11.

In the case of a multicast communication, in addition to each direction, an extra bit indicates if the packet is intended for the current cluster. Thus, the router knows if it must forward a copy of the data on the local link (Wires `L` on Fig. 3.3).

3.3 Memory

The MPPA has three levels of memory: the registers (and the caches) of the PEs, the 2-Mbytes shared memory in each cluster (for 16 PEs), and a Double Data Rate (DDR) memory provided with the board.

The shared memory can be directly accessed from a PE. There is a writing buffer and a cache, but there is no cache coherency mechanism to allow predictable memory accesses latency [6]. Furthermore, the caches of the processor have the LRU cache replacement policy. This policy is predictable and hence, allows to compute a tight bound of the memory access times [8].

To minimize contention the shared memory is composed of 16 banks of 128 bytes each. Hence, it avoids interference between two PEs writing in two different banks.

For the Data-NoC (D-NoC) communications, the new packets are received in the shared memory through a DMA. The incoming buffer is specified when the DMA is configured (before the first reception). For the Control-NoC (C-NoC), the data (limited to 64 bits) is received in a special register. To improve the NoC latency, DMA is prioritized over other memory accesses. Namely, when a packet arrives, it is directly written in memory and can be in conflict with another concurrent access in the same bank (access in other bank is not impacted).

For outgoing packets on the C-NoC, register contains the data, so there is no conflict possible with the memory accesses. For the D-NoC, the DMA reads a (configurable) buffer in the shared memory, and it is not prioritized over the other accesses.

3.4 Individual Cores

Timing anomalies are contra-intuitive influence of the local execution time on the global execution time [12].

The cores of the Kalray's MPPA are designed to avoid timing anomalies. In particular the LRU replacement policy is chosen for the caches (allowing to statically know the memory access latency [8]).

Concerning the pipeline, there is no branch prediction (except for loops, but the penalty is constant [6]), and no out-of-order execution. The VLIW allows to take advantage of the instruction level parallelism in a static way since the scheduling of instructions is done by the compiler [6].

Mesosynchronous Two clocks are mesosynchronous when they have the same period, but there is a constant phase between them.

Each core is featured of two 32-bits timers, that can be chained to have one 64-bits timer. All the timers of the MPPA are mesosynchronous [6].

3.5 Using the Kalray's MPPA for Reactive Systems

Bounding the execution time of a part of a program is a required condition to execute reactive programs on hardware. We call Worst-Case Execution Time (WCET) the upperbound of the execution time. Most of the recent processors have timing anomaly [12] making difficult the computation of an accurate WCET (definition in Section 3.4). In general, all operations of the program must be bounded. Hence, on a many-core architecture, the communications time must also be bounded. We call this bound Worst-Case Transmission Time (WCTxT).

The architecture of the Kalray's MPPA has specificities allowing to statically compute an accurate WCET on each core. The first is there is no branch prediction and complex pipeline. When present, these functionalities lead to timing anomaly. Second, the VLIW instruction parallelism is fully managed at compile time and can be disabled. Third, there is no cache coherency protocol and the cache replacement policy is the LRU one. Furthermore, the shared memory is parted in banks avoiding interference if each core accesses its own bank. For NoC communications, a flow control mechanism makes the latency of the packets boundable and the throughput guaranteed.

All these characteristics make the Kalray's MPPA a good candidate for the deterministic implementation of reactive systems.

— 4 —

Problem statement

4.1 Running Synchronous Programs on Many-core Architectures

Implementing a synchronous program to take advantage of a many-core architecture raises several problems. We describe them in the next sections.

4.1.1 How to Map the Nodes on a Many-core Architecture?

Fig. 4.1-(1) shows an example of system specifications. It gives the sampling rates for the inputs, and the refresh rates for the outputs. Another information is the input-output latency. This information strongly depends on the environment and cannot be changed. From this specification one can deduce the activation period of each node, as shown in Fig. 4.1-(2). We consider the periods as a specification to take into account for mapping of the nodes.

Mapping the nodes on a many-core architecture depends both on the high level description (the activation periods, the wires between the nodes) and on the architecture. The many-core architecture offers several levels of parallelism (instruction-level, core-level, cluster-level). To solve this problem, we have to investigate all these possibilities to find a placement under the constraints of the high level description. Our work does not deal with the problem of placement. We consider that the mapping of the nodes on the processor is given (either manually or by another tool).

We investigate the communication through the NoC because it is required to map a reactive program on a many-core architecture. Using the NoC communications in a reactive system is also something new we want to investigate.

4.1.2 How to Define a Deterministic Semantics of Data Exchanges Between the Nodes?

We need to give deterministic semantics to the communication between the nodes. Namely, we need to find the data-exchanges semantics we can preserve when we map a program onto the many-core architecture. For instance, we showed in Section 2.4.3 that the "freshest value" data-exchange implementation is not deterministic. Hence, we need to find the appropriate semantics. We define the problem in Section 4.2.

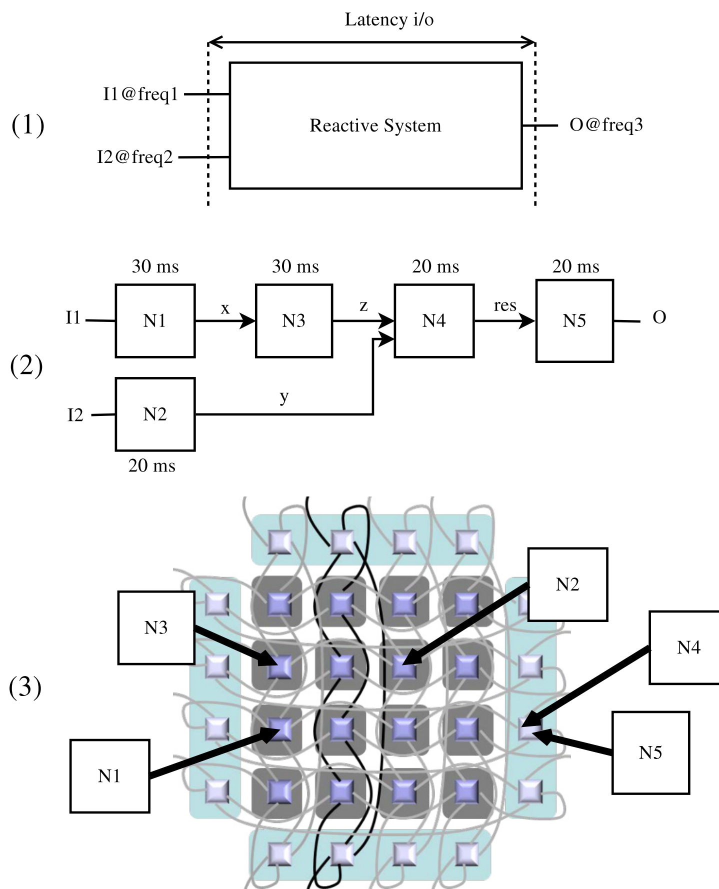


Figure 4.1: Implementation steps of a reactive program on a many-core architecture. There are three steps: the first is given by the specifications of the system. Each input and output has a sampling rate (or a refresh rate for the outputs). The input-output latency of the pairs of input-out is also given. The second step is to deduce, from the first step, the activation period of each node of the system. The third step corresponds to the mapping on the architecture.

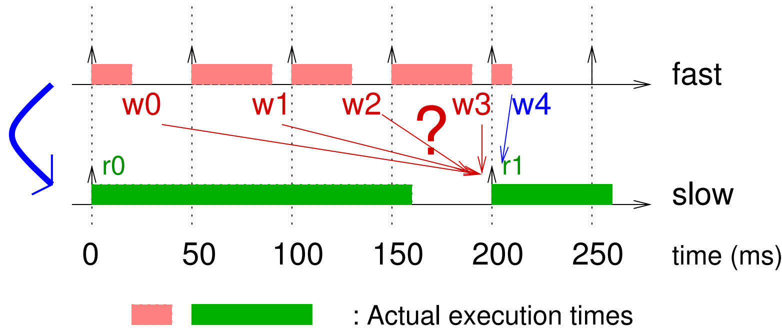


Figure 4.2: The rectangles represent the actual computation time of each task. At the end of each computation, a new value is available. We can choose to send some to the slow task. Values from w0 to w3 are available before the activation r1 of the slow task. Value w4 is available after r1. If we choose this value, the slow task has to wait.

4.1.3 Previous work

The previous work by Hanan Kanso [9] is an implementation of a synchronous program on the Kalray MPPA. It shows that parallelizing a synchronous program on the Kalray MPPA is possible. Each node is mapped on a Resource Manager (cf. Section 3) and communicates through the Data-NoC. The solution works only for nodes with harmonic activation periods. Moreover, it uses the POSIX API of the MPPA which gives less control on the system than the low-level API.

4.2 Problem Definition

Let us consider a reactive program composed of several nodes. The activation periods (which not be harmonic) of the nodes and the mapping on the clusters of the Kalray MPPA are given. We know the WCET of each node of the program. With these conditions, our contribution is to answer several questions:

1. How to define deterministic semantics of communication between these nodes on the high level description?
2. How to preserve the semantics of the high level description in the many-core implementation?

4.2.1 Defining a Deterministic Semantics of Communication

We need to define the semantics of the data-exchanges. For instance in Fig. 4.1-(2), the wires mean that N3 generates values and N4 needs these values, but nothing defines precisely when. We need to define the relation between the inputs and the outputs of the nodes. Example Fig. 4.2 shows, for two tasks, the values we can choose to send from a task to another.

The semantics needs to be deterministic because determinism of a system is a condition to allow the verification of the correctness of the implementation by comparing the implementation with the model. The example of "freshest value" of Section 2.4.3, shows that some data-exchanges implementations are not deterministic.

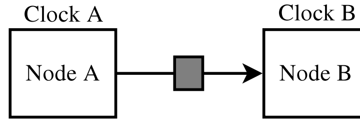


Figure 4.3: The unit-delay semantics connecting two nodes working on different clocks.

```
-- "current" with an initialization value
node generic_current<<type ty>>(c: bool; v: ty; (x: ty) when c)
    returns (o:  ty);
var
    oncec: bool;
let
    oncec = c or (false -> pre oncec);
    o = if not oncec then v else current(x);
tel

node generic_UnitDelay<<type ty>>
    (clockA,clockB:bool; (input:ty) when clockA; val:ty)
returns ((output:ty) when clockB);
var
    currOutput: ty;
let
    currOutput = generic_current<<ty>>
        (clockA, val,(val when clockA) -> pre input);
    output = currOutput when clockB;
tel

node UnitDelay = generic_UnitDelay<<int>>;
```

Figure 4.4: Lustre model of the unit-delay semantics.

Unit-delay semantics We consider a node A generating a flow of values on a clock A. Node B reads one value per activation of a clock B. The read value is the previous value in the output flow of A *i.e.*, the value generated by the previous activation of A when B reads. It is expressible with a `current` and a `pre` operator in Lustre (e.g., `Y = current pre X`). The Lustre code of the `UnitDelay` node is given in Fig. 4.4. The following table gives the behavior of the unit-delay semantics:

Base clock	1	1	1	1	1	1	1	1	1	1	1
Clock of input	1	0	1	0	1	0	1	0	1	0	1
Instants of input	1		2		3		4		5		6
Input	v_0		v_1		v_2		v_3		v_4		v_5
Clock output	1	0	0	1	0	0	1	0	0	1	0
UnitDelay(A)				v_0			v_2			v_3	
Instants of output	1			2			3			4	

Fig. 4.6 shows the Lustre code of a four nodes connected with the unit-delay semantics.

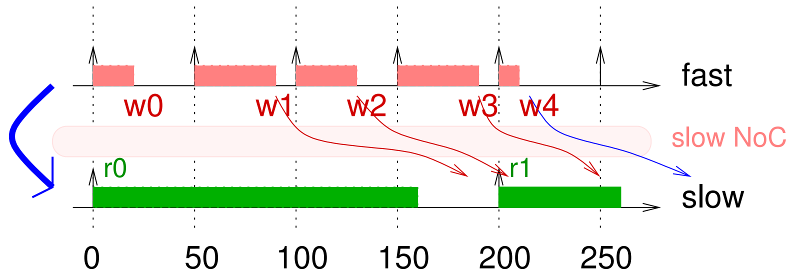


Figure 4.5: The rectangles represent the actual computation time of each task. At the end of each computation of the fast task, a value is sent through the NoC. Due to the speed of the NoC, values w2 and w3 may not be available at activation r1 of the slow task even though they were sent before this instant.

4.2.2 Preserving the Semantics of the High Level Description in the Many-core Implementation

The time of communications through the NoC is not negligible, thus, it is not always compatible with the strong timing constraint required by the instantaneous communication. Fig. 4.5 shows the consequence of the NoC on available values. For example, with the instantaneous communications (we read value w4 at activation r1), the tasks cannot start on their activation because the input values are not available at activation.

In our solution, we choose the unit-delay semantics *i.e.* with at least one writer period of delay (values w3, w2, w1 or w0). On this figure, neither w3 (one unit delay), nor w2 (two unit delays) are available, hence we can choose w1 (three unit delays) or w0 (four unit delays).

The problem is to implement the unit-delay semantics on a many-core architecture. As the NoC introduces communication latencies, we have to verify whether the unit-delay semantics holds when a program is implemented on the actual hardware.

```
node Connection(useless: bool) returns (currX, currY, currRes: int)
var
    seq: int;
    init, highC, lowC: bool;
    (N1, N3: int) when highC;
    (N2, N4: int) when lowC;
    x: int when highC;
    (y, z, res: int) when lowC;
let
    init = true -> false;

    -- Define clocks
    seq = 0 -> ((pre seq + 1) mod 6);
    highC = (seq mod 2 = 0);
    lowC =  (seq mod 3 = 0);

    N1 = Node1(init when highC);
    N2 = Node2(init when lowC);
    N3 = Node3(init when highC, x);
    N4 = Node4(init when lowC, z, y);

    x = UnitDelay(highC, highC, N1, 0);
    y = UnitDelay(lowC, lowC, N2, 0);
    z = UnitDelay(highC, lowC, N3, 0);
    res = UnitDelay(lowC, lowC, N4, 0);

    -- To display wires.
    currX = generic_current<<int>>(highC, -1, x);
    currY = generic_current<<int>>(lowC, -1, y);
    currRes = generic_current<<int>>(lowC, -1, res);
tel
```

Figure 4.6: Example Lustre node connecting four nodes. Node 2 and 4 are activated on clock `slowC`, and Node 1 and 3 are activated on `highC`. The four nodes are connected with the unit-delay semantics. For instance, the wire `z` connects the output of `N3` on clock `highC` with `N4` on clock `lowC`. Nodes `generic_current` and `UnitDelay` are given in Fig. 4.4. Complete code is given in Appendix A.

— 5 —

The unit-delay semantics: deterministic and preservable data-exchanges between nodes

We choose the unit-delay semantics for the data-exchanges. To preserve the semantics on the many-core architecture, we statically compute patterns defining for each node, when it sends, and when it receives the values.

5.1 Definition of a Task

The reactive program is split into tasks that are mapped on the architecture.

Let T_i be a strictly periodic task, namely we consider its deadline equal to its period.

A task T_i is defined by $(p_i, WCET_i)$. With p_i a period and $WCET_i$ the worst case execution time of the task. We always assume $p_i \geq WCET_i$. $r_{i,n} = n * p_i$ represents the n*th* activation of the task T_i. The first activation of the task T_i is $r_{i,0}$.

$O_{i,n}$ represents the output computed at $r_{i,n}$ by the task.

5.2 Communication Pattern

The purpose of this section is to reproduce the exact behavior of the unit-delay semantics using communications on the NoC. Fig. 5.1 gives an example of two tasks communicating at different speeds with the unit-delay semantics.

We consider two tasks with different periods (both periods are defined on the base clock of the system). The code of a task is executed at each activation, hence on the period of this task. At each activation, the code must decide if it communicates (in other case, the reader keeps the old value, and the writer does not send the value). When two tasks are communicating, the fastest task is responsible for the communication:

- If the fastest reads, and the slowest writes, the fastest reads only sometimes but the slowest writes every time.
- If the fastest writes, and the slowest reads, the fastest writes only sometimes but the slowest reads every time.

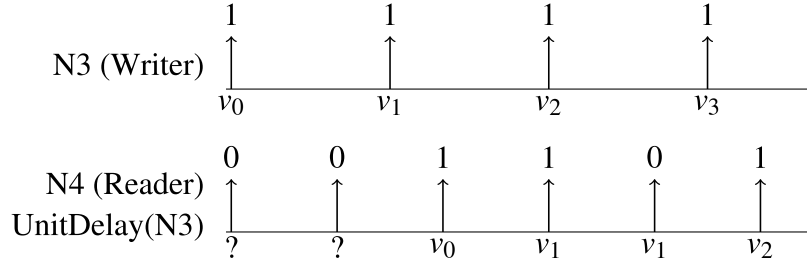


Figure 5.1: At each activation, the '1' on the arrow indicates if the task must communicate (write for N3, read for N4). N3 writes at each activation. According to the unit-delay semantics, N4 needs to read only at the 3rd, 4th and 6th activations. Hence, the pattern begin with '0's. At each '0', the previous value is kept (or the initialization value for the first activation).

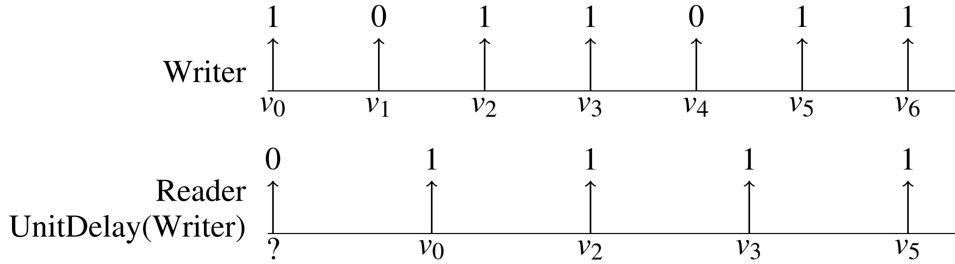


Figure 5.2: At each activation, the '1' on the arrow indicates when the task must communicate. In this example, we demonstrate that sometimes a slow reader needs an initialization, namely, Reader does not read in the first activation because the v_0 is not yet available. Hence, the eader must use an initialization value for the first activation, and, due to the initialization, it need a pattern.

In both cases, we can justify it because, for two communicating tasks, there is always one activation of the fast task between two activations of the slow one. Hence, we are sure that: the fast reader has consumed the value, or the fast writer has produced a value. Note that, for the slow reader, an initialization is needed in most of the cases.

In a pair of communicating tasks, we need a pattern for the slower and a pattern for the reader indicating when these tasks communicate. As shown in Fig. 5.1, the fast task does not read at every activation and sometimes keeps a value for several activations (communication-by-sampling). This figure also gives an example pattern. Furthermore, in Fig. 5.2, we show that the slow task also needs an initialization and hence needs a pattern.

$Pattern(k,n)$ indicates, for the activation $r_{k,n}$, if the task T_k must communicate, *i.e.*, depending of the nature of the task, if it must read or write. T_k must communicate at activation $r_{k,n}$ iff $Pattern(k,n)$ is 1.

The patterns are ultimately periodic as defined in [4]. A pattern is a word on the alphabet $\{0,1\}$ of the form $u(v)$. It is composed of an initialization u and a periodic part v (the periodic part is repeated infinitely). We give the grammar of a pattern:

$$u ::= b*$$
$$v ::= b.b*$$

$$b ::= 0 \mid 1$$

Example, the two patterns of Fig. 5.2 are: (101) for the writer $0(1)$ for the reader.

The size of the periodic part v depends on the least common multiple of the two periods (hence, we consider the periods are rationals numbers):

$$sizePeriodic_i = |v_i| = \frac{lcm(p_i, p_j)}{p_i}$$

5.3 Computing Communication Patterns

```
Input: The periods pw and pr, and the number of unit delays k
1  function writerActivation(r_{j,n}, k) = ⌈(r_{j,n} − (r_{j,n} mod p_i) − k*p_i) / p_i⌉
2  function sizeInit(k) = ⌈(k*p_w) / p_r⌉
3  sizeInitR ← sizeInit(k)
4  sizePeriodicR ← lcm(pw,pr) / pr
5  sizePeriodicW ← lcm(pw,pr) / pw
6  array pattW [0, pw − 1] of Boolean values
7  array pattR [0, sizeInitR + sizePeriodicR − 1] of Boolean values
8  lastUnitDelayW ← -1
9  for a ∈ [0, sizeInitR + sizePeriodicR[ do
10     unitDelayW ← writerActivation(a, k)
11     if unitDelayW ≥ 0 and lastUnitDelayW ≠ unitDelayW then
12         pattR [unitDelayW ] ← 1
13         pattW [a ] ← 1
14     end
15     lastUnitDelayW ← unitDelayW
16 end
17 return (pattW, pattR)
```

Algorithm 1: This algorithm computes the communication patterns reproducing the behavior of the unit-delay semantics. It produces two patterns one for the reader and one the writer. The first sizeInitR elements of pattR is the initialization part, the rest is the periodic part. PattW does not contain initialization.

For a pair of communicating tasks (a writer T_w, and a reader T_r), we compute two patterns: pattW et pattR. The principle of this algorithm is:

- For each activation of T_r, it computes the corresponding activation of T_w according to the unit-delay semantics using the function *writerActivation*. Namely, in Fig. 5.3, at the second activation, Reader gets v_0. The corresponding activation is the first activation of Writer.
- In the writer pattern pattW, it marks the corresponding activation (T_w must send the value). (L12)
- In the reader pattern pattR, we also mark the activation, but only if it is the first time the reader can read the value (because T_w writes the value only one time). (L13)

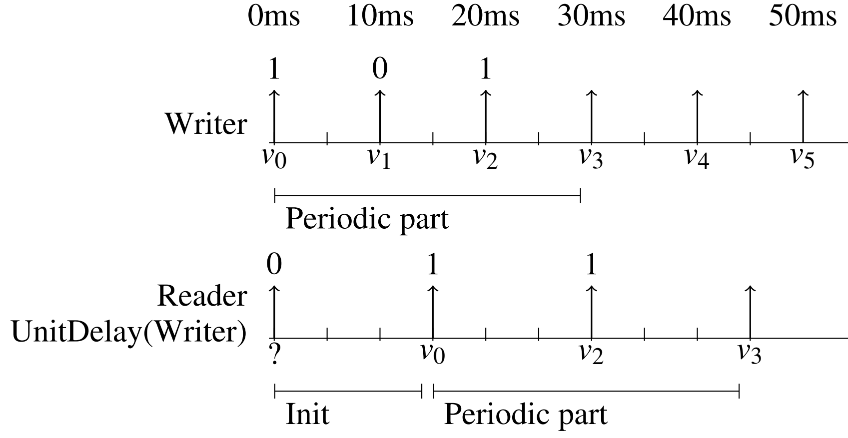


Figure 5.3: The pattern of Reader is composed of two parts: an initialization part (Init), and a Periodic part. The pattern of the Writer is only composed of a periodic part. The algorithm computes the Writer and the Reader patterns by evaluating, for each activation of Reader, which value of the Writer is needed (according to the unit delay semantics).

For the activation $r_{j,n}$ of a reader T_j, the function *writerActivation* gives the activation of the writer T_i when the value is produced. k is the number of unit delays.

$$writerActivation(r_{j,n}, k) = \left\lceil \frac{r_{j,n} - (r_{j,n} \bmod p_i) - k * p_i}{p_i} \right\rceil$$

The complete algorithm is given in Algo. 1.

5.3.1 Initialization of the Reader

In Section 5.2, we showed that sometimes the reader needs an initialization because during its first activations, no value of the writer has been received yet. For instance, in Fig. 5.3, v_0 is available at the second activation of T_r, hence, the initialization part (Init on the figure) lasts one period of Reader.

We define the initialization as the number of periods of the reader needed to have the first value of the writer according to the unit-delay semantics. Function $sizeInit(k)$ gives the size of the initialization of the reader T_j.

$$sizeInit(k) = |u_j| = min\{n | writerActivation(r_{j,n}, k) = 0\} = \left\lceil \frac{k * p_i}{p_j} \right\rceil$$

For a unit delay of k, the value is read at least k periods of the writer after. Hence, we compute the first activation of the reader after these k periods.

5.4 Verifying the Availability of Values

The constraint of our solution is: *the input values must be available before the activation of the task*. We need to guarantee this property even if the Worst-Case Transmission Time (WCTxT) is high. Fig. 5.4 depicts a case when the WCTxT is too high. Reader cannot start since the input v_1 is not available at activation.

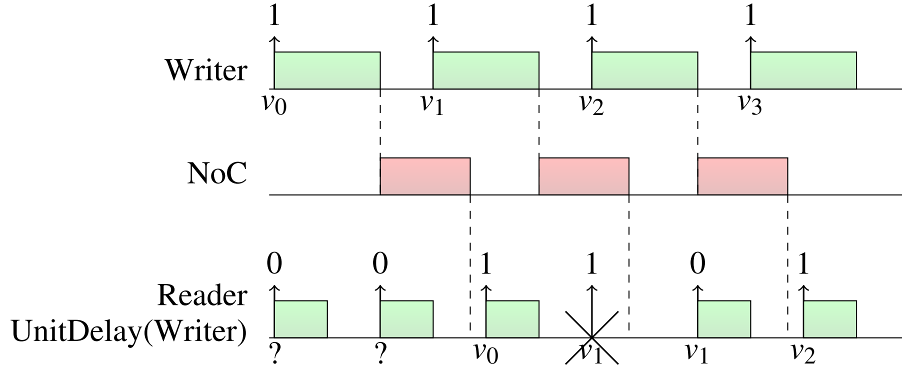


Figure 5.4: We represent the WCET of the tasks Writer ($p_w = 3$, $WCET_w = 2$) and Reader ($p_r = 2$, $WCET_w = 1$) connected with one unit delay. The line NoC represents the transmission through the NoC, the rectangles are the Worst-Case Transmission Time (WCTxT = 2). The 4th activation of Reader needs v_1 but due to the high WCTxT it arrives after the activation.

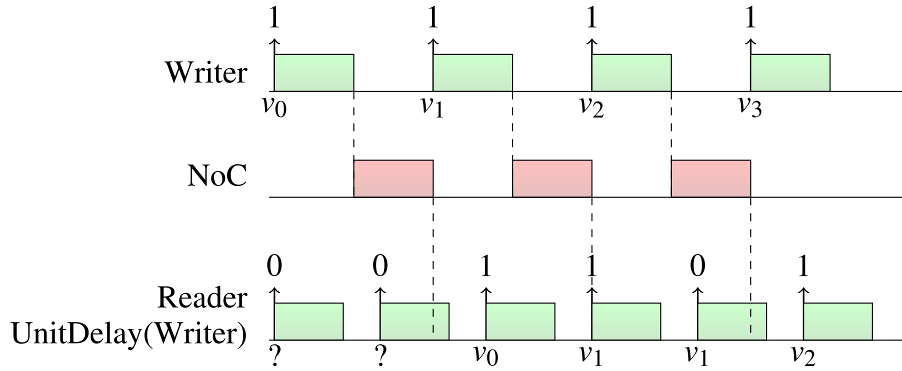


Figure 5.5: In this example the WCET of 1.5 and the WCTxT of 1.5 both fit into the period of the writer ($p_w = 3$). In this case, the communication is guaranteed to work.

We first consider a simple case, when we include the transmission time in the period of the writer. Thereafter, we consider the general case.

5.4.1 Simple case $p_w > WCET_w + WCTxT$

To simplify the problem, we consider that the period of a task T_w is enough to compute and send the data in one period. Namely: $p_w > WCET_w + WCTxT$.

Fig. 5.5 shows a case when the WCTxT and the WCET both fit into the period. In this case, Reader is guaranteed to receive the data before his activation. The worst case is when Writer sends v_1 because Reader needs it exactly p_w after. As $p_w > WCET_w + WCTxT$, T_r is guaranteed to receive the value. As a consequence, one unit delay is always sufficient when $p_w > WCET_w + WCTxT$.

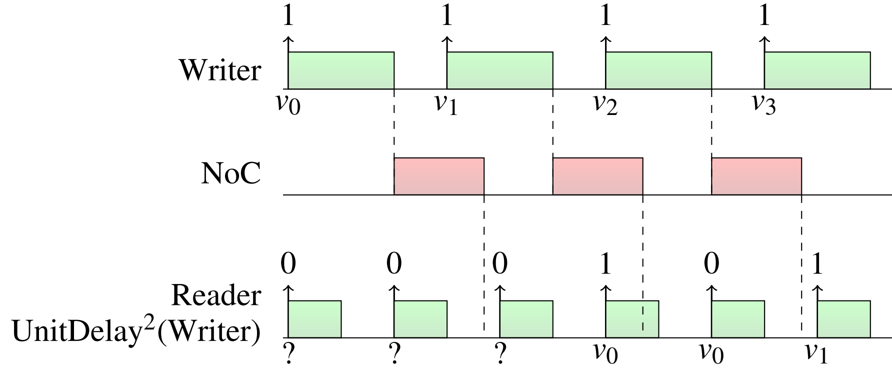


Figure 5.6: This figure shows the same example as Fig. 5.4 with a two unit-delays semantics. With this semantics the data is always available.

5.4.2 General case $p_w > WCET_w$

The general case is useful if we cannot accelerate the computation to make $WCET_w + WCTxT \leq p_w$. As a consequence, we sometimes need several unit delays to guarantee the availability of the data before the reader needs it.

In Fig. 5.4, the input data is not always available before the activation. If we change to the two-unit-delay semantics, it works. The same nodes with this semantics are represented in Fig. 5.6.

The function *minUnitDelay* gives, for a Writer and the WCTxT, the minimal number of unit delays. We can see the number of unit delay as the number of periods of the writer able to compensate a high WCTxT.

$$minUnitDelay(Tw) = \left\lceil \frac{WCET_w + WCTxT}{p_w} \right\rceil$$

Example, for $p_w = 3$, $WCET = 2$, $WCTxT = 2$, the minimal number of unit delays is 2.

5.5 Communication Buffer

In example Fig. 5.7, Writer can send two values before Reader receives them, hence we need a communication buffer. The reason is data are sent once they are produced but they are read only when Reader needs them. In this part we give an algorithm to compute the size of this buffer.

5.6 Buffer Size

The n-synchronous theory [4] defines the adaptability that tells whether a communication buffer can be bounded. Another result of this theory allows to compute the size of this buffer.

Still, to be able to apply the n-synchronous theory, the patterns must be all expressed on the same clock. We choose to express all the patterns on a period of $gcd(pw, pr)$. Example, the writer patterns (1) of period 3 becomes (100), and the reader pattern 00(110) of period 2 becomes 0000(101000) (the common period is $gcd(2,3) = 1$).

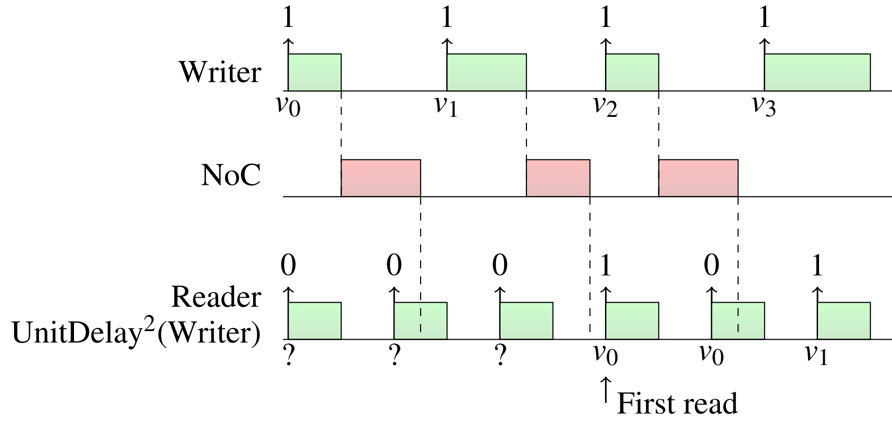


Figure 5.7: In this figure we represent some possible execution time and transmission time instead of the WCET and WCTxT. At the first read, v_0 and v_1 have already been sent. As a consequence, we need a buffer to receive the values.

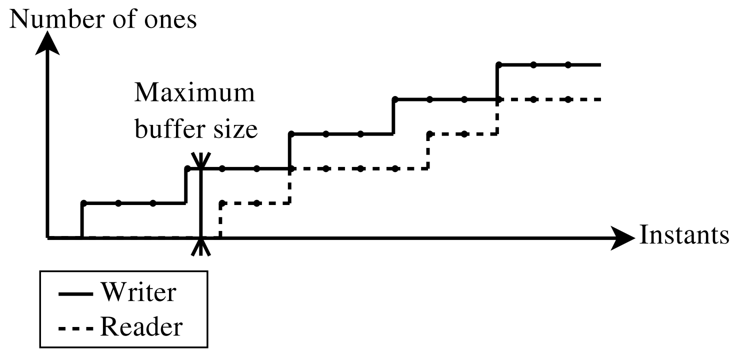


Figure 5.8: A writer ($p_w = 3$) of pattern 1001(001001) is sending to a reader ($p_r = 2$) of pattern 0000(101000). The maximum buffer size is given by the maximum difference between the number of written and the number of read values.

Adaptability (written $w1 <: w2$) [4] The communication ensures a task never reads an empty buffer and this buffer can be bounded. In other word, the data produced on a rhythm $c1$ can be consumed on a rhythm $c2$ by insertion of a bounded buffer.

The unit-delay semantics ensures we read only an already produced value, hence we never read in an empty buffer and our algorithm guarantees all the produced values are read (same number of one in the two periodic parts) hence the buffer can be bounded. Hence, the adaptability of our patterns is guaranteed.

According to the n-synchronous theory [4] the size of the buffer is the maximum difference between the number of written data and the number of read data in the buffer (as depicted in Fig. 5.8). We can compute this step by step. The n-synchronous theory gives the bound of this computation:

For ultimately periodic patterns, if $w1 <: w2$, the maximum buffer size is met before the period $P = max(|u1|,|u2|) + lcm(|v1|,|v2|)$.

Algo. 2 computes the minimal size of the buffer able achieve the communication.

```
Input: pw, pr, delay and init. pattW, pattR must be expressed on lcm(pw,pr).
1 nbWrites ← 0
2 NbReads ← 0
3 maxBuff ← 0
4 for i ← 0 to init + lcm(pw, pr) do
5 |   if pattW[i] then nbWrites ← nbWrites + 1
6 |   if pattR[i] then nbReads ← nbReads + 1
7 |   maxBuff ← max(maxBuff, nbWrites- nbReads)
8 end
9 return maxBuff
```

Algorithm 2: This algorithm computes the maximum buffer size. The patterns must be both expressed on the clock lcm(pw,pr).

— 6 —

Implementation of a multi-periodic synchronous program on the MPPA

Extracts of the implementation presented in this section is given Appendix B.

6.1 General Principles of Implementation

Each cluster of the Kalray is composed of 16 Processing Elements (PE) and 1 Resource Manager (RM). In the clusters we choose to implement the nodes on the RMs because they are designed for NoC communication. Sending packets from the PE is possible, but requires more configuration (the default configuration forbid access to DMA from the PEs).

To load a program on the MPPA, we load a binary containing several executables: one executable for the I/O cluster and the executables to be loaded on the clusters. The executable for the I/O cluster is launched first. It spawns the executable on each Resource Manager (RM).

6.2 Case Study

We give an implementation for the system example shown in Fig. 6.1. The principle of our solution is to map each node on a cluster. The wires x, y, z and res are communication through the NoC with the unit delay semantics. Nodes N1, N2, N3, N4, N5 are respectively mapped on RMs 3, 4, 1, 8, 10. We use I/O Cluster 128 to spawn the executables on the Resource Managers.

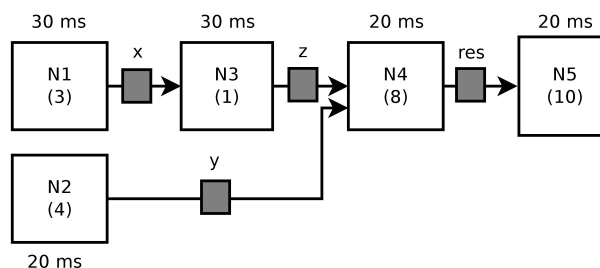


Figure 6.1: Mapping of the nodes of our solution on the Resource Managers of the Kalray MPPA. The numbers in parentheses are the cluster identifiers on the MPPA Gray squares represent wires with the unit-delay semantics. Example Lustre code for this system is given in Fig. 4.6.

6.3 Low Level Communications on NoC

When a node sends data to another node (for instance to implement the links x, y, z and res on Fig. 6.1), the data is sent through the NoC. A route is embedded in the header of the packet to indicates the direction to take, at each router, to reach the destination.

The data is received in a buffer. As a node can have several buffers (to receive several kinds of data), we must assign them a tag. Hence, to send data to a node, we must know, the destination cluster, the route and the tag of the buffer.

There are several steps for the implementation of the communication between the nodes:

- For each reader: create the reception buffers, and assign a reception tag to the buffer.
- For each writer: compute the route and define the packet header.

We give the implementation details in the next sections.

6.3.1 Reader side

Configuration of the reception

The configuration of the reception and the activation of the reception are done using two low-level procedures.

```
_k1_dnoc_configure_rx(recvTAG, _K1_NOCV2_INCR_DATA_NOTIF,
    &recvBuffer, sizeof(recvBuffer), 0);
__k1_dnoc_activate_rx_ext(IFCE, recvTAG);
```

`RecvTAG` is the reception tag. This integer is useful for the DMA to know to which buffer the data is addressed because it is possible to receive data in several places. Hence, the writer must also know the tag of the buffer. `_K1_NOCV2_INCR_DATA_NOTIF` is used to ask the DMA to generate an event when a new packet is received. We use it to increase security, since the principle of our solution guarantees that data is arrived when we read them. `IFCE` is to select the port, as RM only have one DMA, this value is always 0.

The configuration is done once, before the first reception.

Cache Coherency Problem

Because of the absence of hardware cache coherency mechanism in the Kalray, we cannot read directly the received data. When a packet is received, the DMA writes in a buffer located in the shared-memory. As there is no cache coherency mechanism and as each RM has his own cache, reading directly the buffer leads to bad results.

Our solution is to use the `ldu` (load double uncached) instruction allowing to bypass the data cache and read a 64-bits (double) value directly in the shared-memory. Kalray provides `__k1_umem_read64` which is an intrinsic function for this instruction.

Read the received data

We can read the data using `__k1_umem_read64(recvBuffer)`:

```
while (!__k1_dnoc_get_notification_counter_rx(recvTAG))
        ;
recvBuffer = __k1_umem_read64(recvBuffer);
__k1_dnoc_event_cntr_load_and_clear(recvTAG);
```

The while-loop is to ensure the data is present before we read: it waits for a new DMA event. The `__k1_dnoc_event_cntr_load_and_clear` clear the event to be able to wait for the next one. This is only defensive programming, namely if the execution times of the programs match the WCET and the core are correctly synchronized, the loop is not executed since the event is already here.

6.3.2 Writer side

Configuring the transmission consists in setting the packets header which contains the information about the routing and the destination. We give below its contents and the values we used in the solution.

Field	Value	Description
Valid	true	The packet is valid.
Multicast	false	One-to-one communication
Tag	readerTag	Destination tag
End of transmission	true	Ask the reader-DMA to generate an event for this packet
Route	routeLSB	21 least significant bits of the route
Protocol	0	protocol: bit extension
Extended	true	enable the bit extension
Routex	routeMSB	most significant bits of the route

The destination tag is used address the right reception buffer. We define this tag in Section 6.3.1. The field End of transmission indicates that the packet comes alone. In our implementation, all the packets are independent. This field is used reader-side to generate a new event for each received packet.

The field Route must contain the route the packets will take. As 21 bits are not always sufficient, the field Protocol with the field Extended allow to extends the header to carry the rest of the route (Routex). In this case, the size of the header is increased. The route is computed statically using the tool `k1-nocencoderoute` provided by Kalray. For a list of nodes (more details in Section 3.2), this tool gives an integer value corresponding to the route. We affect field Route with this integer.

```
__k1_dnoc_configure_tx(IFCE, Tag, firstDirection, header, bandwidth);
```

The code above gives the configuration of the output. It must be done once, before the first send and takes the header in parameter. `Tag` is a local identifier for the transmission, and must not be confused with the destination tag. `firstDirection` is the first direction of the route. It is given by Kalray's tool `k1-nocencoderoute`. `Bandwidth` lets us specify the maximum bandwidth (cf. Section 3.2) for flow control (0 means best effort).

```
__k1_dnoc_send_data(IFCE, Tag, &o, sizeof(o), SEND_EOT);
```

This code sends a packet with the configuration identified by `Tag`. Variable `o` is the output buffer. `SEND_EOT` ask the DMA to send the packet now, without grouping several packets.

6.4 Patterns

We give the code to implement the patterns. It is the core of the unit-delay semantics reproduction. It implements the test to know if the entity must send/receive the data according to the pattern. This code is given when both patterns are composed of an initialization and a periodic part but we can simplify it if one part is constant.

```
#define PATTERN_INIT 2
#define PATTERN_PERIOD 3
#define PATT_ACTIVE(n) (1&(pattern>>(PATTERN_PERIOD+PATTERN_INIT-1-n)))
uint64_t pattern = 0b10110; // Pattern 10(110)

...
// Computation, Lustre step, etc
...
if (PATT_ACTIVE(period)) { /* Send, or receive */ }
period++;
if(period == PATTERN_INIT + PATTERN_PERIOD) { period = PATTERN_INIT; }
```

6.5 Implementation of the Non-Drifting Period

Each core (regardless of the kind: PE, RM or I/O Cluster) owns two 32-bits timers. A 64-bits counter can be created by chaining them. As all the clocks of the MPPA are mesosynchronous (cf. Section 3.4), they are decremented synchronously on each core.

We read the timer during the first instant and we compute the beginning of the next instant ($next = t + p_i$). After the execution we wait for the next activation:

```
while(readTimestamp() < next)
        ;
```

With this solution, the activation instants depend only on the first activation, hence the periods do not drift.

We use the intrinsic `__k1_timer64_setup` to configure the timer, and `__k1_timer64_get_value`, to read its value.

6.6 Spawning of the Nodes on the Clusters

When we load a program on the MPPA, it is loaded on an I/O Cluster. This program is responsible to launch (or spawn) the sub-programs on the RMs. During this thesis, we only have access to an early version of the Kalray low-level API (the version 1.3). The low-level spawn is not available. Hence, we use the POSIX spawn in our code.

```
int pid1 = mppa_spawn(clusterID, NULL, "cluster_executable1", argv, 0);
```

`ClusterID` corresponds to the identifier of the cluster where we launch the executable (1, 3, 4, 8 or 10 in Fig. 6.1). The string `"cluster_executable1"` is the name of the cluster program object file. The array argv is to pass arguments to main of the cluster program.

Note, it is not possible to mix low level and POSIX library on the same core. Hence, the low level communication between a POSIX core and a low level core is not possible.

6.7 Programs Synchronization

A sequential code is responsible of spawning the executables on the MPPA (see Section 6.6) hence all the programs do not start at the same time. According to our measurements, the order of magnitude of the duration of one spawn is about 300 μs, hence if we spawn three programs, we possibly have a phase of $2 \times 300 = 600$ μs between the first and the third. The actual duration depends on the size of the program and the network traffic.

If the writer writes too late, *i.e.*, the first activation of the writer was after the first activation of the reader, the reader must wait for the data. On way to solve this problem could be to included the phase in the Worst-Case Communication Time. In the other way, *i.e.* the reader reads too late, it could increase usage of the buffer.

Our synchronization solution is to use the counters of the Debug Support Unit (DSU) of the Kalray. There is one counter per cluster. At the beginning of each program, we wait for a specific timestamps (greater than the time it takes to spawn all the programs). According to the documentation, as the counters are initialized through the NoC, there can be a difference of about 25 μs between the clocks. For an actual hard real-time implementation we must exactly synchronize the cores, but it is out of the scope of our work.

— 7 —

Evaluation

If a Lustre model gives the semantics of reference for the implementation, by transitivity it guarantees the equivalence between all the executions of the implementation. Hence, it proves the determinism. But, to prove the correctness of the solution formally, we would need to model the MPPA. As a master thesis is too short to carry out this modeling task, we choose to evaluate our solution by tests. Namely, we set up a validation architecture. The purpose is to find potential errors in the correction of the implementation or in hardware.

7.1 Reference

The reference is the Lustre implementation of the synchronous program of the case study presented in Section 6.2. The full Lustre listing for one test is given in Appendix A and Fig 7.2 gives a glimpse of the computation of each node. All the links have the unit-delay semantics.

Fig. 7.4 gives the timing diagram of a simulation of a reference synchronous program. We compare this result to the many-core implementation.

7.2 Variation on Sources of Nondeterminism

To show that our solution is deterministic, we need to show that all the executions of the implementation are equal. To improve the tests, we exagerate the sources of nondeterminism we could meet in an actual usage.

The two sources of nondeterminism we highlight are the computation time and the transmission time over the NoC. Namely, we must add in the implementation some mechanisms to make them vary.

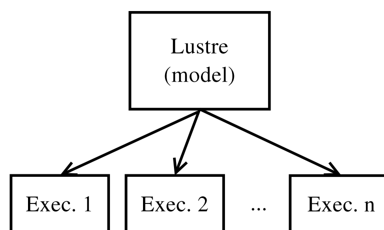


Figure 7.1: If the implementation is equivalent to the reference semantics (written in Lustre). All the executions are equivalent, and hence, the execution is deterministic.

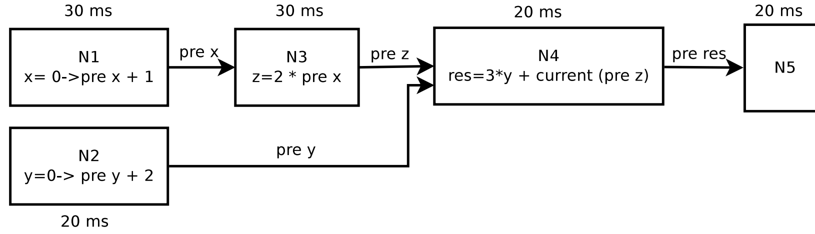


Figure 7.2: Reference system (Lustre code in Appendix A)

7.2.1 Variation of Computation Time

The simulation of the computation time is done by adding a random duration after the actual computation. It simulates an actual computation since the processor is actually running. One could argue it does not simulate actual computation since it does not access memory (except to read timestamps). But, as all the cluster memories are isolated we can consider that a more advanced simulation is not necessary.

```
duration = rand()%MAX_RAND_TIME;
uint64_t initial = readTimestamp();
while(readTimestamp() - initial < duration)
    ;
```

`MAX_RAND_TIME` is the maximum time the program can wait without missing the next activation. We obtained this value by measuring (with timestamps) the duration of the computation of the nodes and the duration of the transmission (a better solution is possible if we know the WCET of the program). `MAX_RAND_TIME` corresponds to the slack time (For a task i, $p_i - WCET_i - WCTxT_i$).

7.2.2 Variation of Transmission Time

To influence the transmission time, we create interferences on some wires. We make a cluster sending data through a link used for the implementation (for instance through the link from R3 to R1).

We add two nodes to the implementation. The first has an integer array (some dozen of 64-bits integers). In an infinite loop, it sends this array and waits a random time. The second ensure the array is received.

7.3 Performance Evaluation of the Low Level Communications

In this section, we assess the performance of the low level communications. We measure the communication latency to have the order of magnitude of the Worst-Case Transmission Time. It is useful to know which kind of program we can distribute and which transmission periods we can hope to have.

Between two Resource Managers, we measure the time it takes for the return trip of 64 bits of payload. We vary the number of hops between the RMs. Fig. 7.3 gives the result. We see that the consequence on the return trip of one more hop is about 50 ns which is low compared to the communication time with 1 hop.

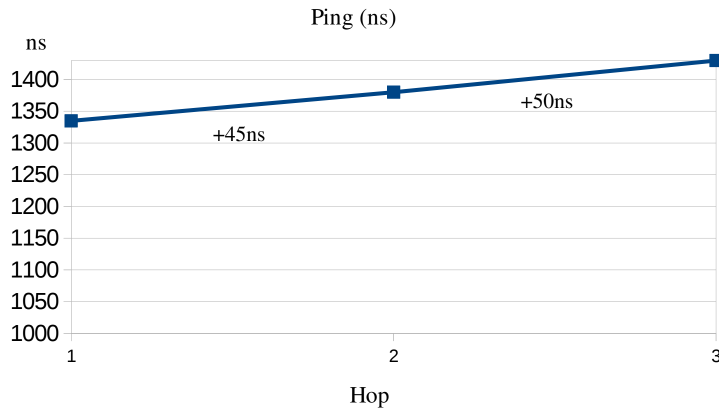


Figure 7.3: Low-level communication performances. Measure of the ping latency in milliseconds. The figure gives the return trip for 64 bits of data for one, two or three hops (one hop mean a communication between two adjacent routers).

The order of magnitude of the return trip is 1.5 μs for 3 hops. We think these figures are promising for reactive systems implementation.

7.4 Results and Conclusion

We presented a test architecture for implementations using our method. We give an example of test for the case study presented in Section 7.1. We compare the reference (Fig. 7.4) to the output of the implementation (Fig. 7.5). The values are coherent with the simulation, and we did not found bugs but we are doing more experiments. We also carried out several experiments with several clocks.

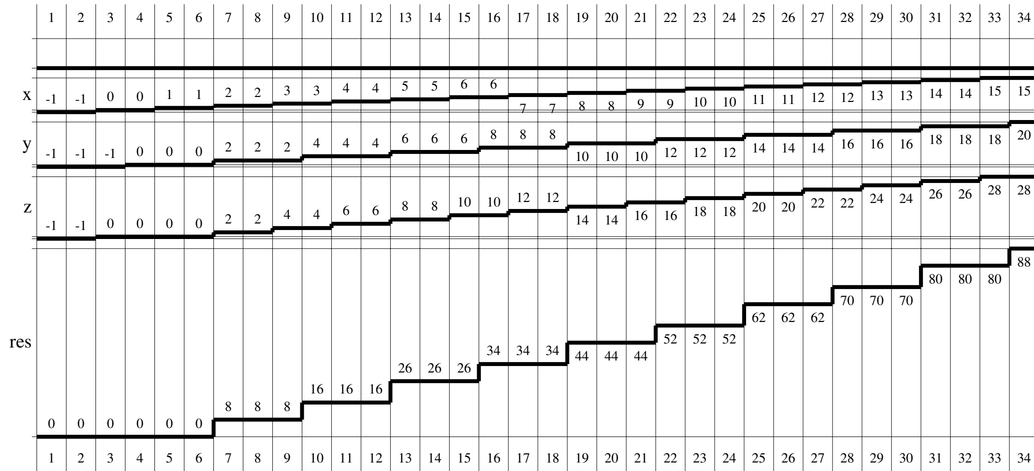


Figure 7.4: Simulation of the Lustre reference synchronous program of Appendix A.

```
...
C1:  (N3) pre x=0, z=0
C10: (N5) pre res=0
C8:  (N4) y=0 , pre z=0, res=0
C1:  (N3) pre x=0, z=0
C10: (N5) pre res=0
C8:  (N4) y=0 , pre z=0, res=0
C1:  (N3) pre x=1, z=2
C1:  (N3) pre x=2, z=4
C10: (N5) pre res=0
C8:  (N4) y=2 , pre z=2, res=8
C1:  (N3) pre x=3, z=6
C10: (N5) pre res=8
C8:  (N4) y=4 , pre z=4, res=16
C1:  (N3) pre x=4, z=8
C1:  (N3) pre x=5, z=10
C10: (N5) pre res=16
C8:  (N4) y=6 , pre z=8, res=26
C1:  (N3) pre x=6, z=12
C10: (N5) pre res=26
C8:  (N4) y=8 , pre z=10, res=34
C1:  (N3) pre x=7, z=14
C1:  (N3) pre x=8, z=16
C10: (N5) pre res=34
C8:  (N4) y=10 , pre z=14, res=44
C1:  (N3) pre x=9, z=18
C10: (N5) pre res=44
C8:  (N4) y=12 , pre z=16, res=52
C1:  (N3) pre x=10, z=20
C1:  (N3) pre x=11, z=22
C10: (N5) pre res=52
C8:  (N4) y=14 , pre z=20, res=62
C1:  (N3) pre x=12, z=24
C10: (N5) pre res=62
C8:  (N4) y=16 , pre z=22, res=70
C1:  (N3) pre x=13, z=26
C1:  (N3) pre x=14, z=28
C10: (N5) pre res=70
C8:  (N4) y=18 , pre z=26, res=80
C1:  (N3) pre x=15, z=30
C10: (N5) pre res=80
C8:  (N4) y=20 , pre z=28, res=88
C1:  (N3) pre x=16, z=32
C1:  (N3) pre x=17, z=34
C10: (N5) pre res=88
C8:  (N4) y=22 , pre z=32, res=98
C1:  (N3) pre x=18, z=36
...
```

Figure 7.5: We give the output of the implementation. Each line is composed of the cluster number, the executed node (in parentheses) and the contents of local values. For instance, the value of `res` is given by C8. As C8 sends the value to C10, C10 displays `pre res`.

— 8 —

Related Work

First, we present some synchronous languages. Second, we present some solutions to run synchronous programs on multi or many-core architectures.

8.1 Specification Languages

8.1.1 Prelude

Prelude is an Architecture Description Language. The purpose is to assemble multi-periodic nodes. These nodes are programmed in C or Lustre. The Lustre clocks are Boolean clocks that are often hard to read for the programmer. Prelude uses strictly periodic clocks instead. They are defined with a period and a phase. It is a sub-class of the Boolean clocks and makes the analysis easier for the compiler (and thus it allows some optimizations).

We present an example of code (from [5]). As the syntax of Prelude is close to the syntax of Lustre, we only talk about the differences.

```
imported node tau1 (i0: int, i1: int)
    returns (o1: int, o2: int) wcet 5;
imported node tau2 (i0: int)
    returns (o1: int ) wcet 10;
imported node tau3 (i0: int, i1: int)
    returns (o1: int) wcet 20;

node sampling (i: rate (10,0))
    returns (o1, o2)
    var vf, vs;
let
    (o1, vf)=tau1(i, (0 fby vs)*^3);
    vs=tau2 (vf/^3) ;
    o2 = tau3 ((vf~>1/10)/^6, (vs~>1/30)/^2);
tel
```

Nodes `tau1, tau2` and `tau3` are imported nodes. They can be executable code for which we know only the inputs, the outputs and the WCET. The clock of the node `sampling`, calling these nodes, has a period of 10 because its only input has a clock (10,0).

Expression `vf/^3` divides the clock frequency of `vf` by 3. Hence, if the clock of `vf` is (10,0), the clock of this expression is (30,0). The converse is the operator `/*`.

Expression `vf~>q` can be compared to a `pre` operator in Lustre, but it is parametrized by `q`, the number of periods of delay.

The operator `0 fby vs` is the initialization operator. It corresponds to `0->(pre vs)` in Lustre.

We give more details in 8.2.1.

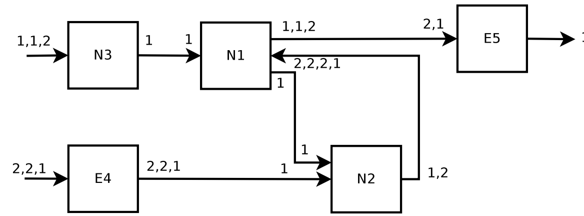


Figure 8.1: Principle of Cyclo-Static Data Flow: for a given node, with each input and each output is assigned a list (output list of N2 is `1,2`, one input list of N1 is `2,2,2,1`). A list give, for each activation, the number of produced or consumed data. These data act as tokens required to activate the node. For instance, between N2 and N1, at the first step, N2 produces one data, and N1 needs two, hence, N2 is executed two times. At the second step, N2 produces two values, and N1 needs one. The lists are repeated infinitely, *i.e.*, if the size of the list is N, an element of the list gives the produced/consumed data every N activations.

8.1.2 ForeC: a Synchronous and Parallel Language

ForeC (presented in [13]) is a synchronous language designed for parallelism and multi-core implementation. It is an extension of C. To be able to guarantee determinism, the language is based on a safety-critical subset of C.

A step in ForeC is composed of three parts. First, the inputs are sampled. Second, each thread computes. Third, the outputs are written. The `par` instruction allows to launch several functions in parallel. The `par` instruction terminates when all the functions finished (join). Each thread finishes a step when it executes the `pause` instruction. The next step begins when all the threads have finished the step.

This language is interesting because it allows a thread-safe way to share variables between the threads. A shared variable declaration comes with a combination function (written in C).

```
shared int total=0 combine with plus;
```

At the beginning of each step, each thread receives a copy of the shared variables. At the end of the step, the local versions are combined with the function and the new value of the shared variable is available for the next step.

8.1.3 Cyclo-Static Data Flow

Synchronous data-flow languages such as Lustre and Prelude are languages based on communication-by-sampling. On the contrary, the Cyclo-Static Data Flow languages are designed to avoid data loss. Each node knows the number of needed values in input and the number of generated values for one execution. The Cyclo-Static Data Flow compute a kind of scheduling to avoid data loss.

As depict Fig. 8.1, in CSDF [1] the constraint is specified in number of received data, and the number of produced data. This way, the data are considered as tokens. The compiler computes (if possible) a schedule avoiding starvation.

8.2 Synchronous Programs on Multi- or Many-core Architectures

8.2.1 Simulation of Distributed Synchronous Programs in Shared Memory

SchedMCore from Mikel Cordovilla [5] is a simulator of many-core platforms for Prelude programs. The input is a set of tasks whose we know the WCET, the execution period, the phase and a binary file (from C, Lustre, etc). We also have the data-dependency words between the nodes. These words can be generated from a Prelude description.

Tasks are communicating through a buffer in shared memory. For a pair of communicating tasks, the data-dependency indicates when the writing task must write and in which cell of the buffer, and when the reading task must read and where. Words are ultimately periodic patterns (parenthesized part is repeated infinitely).

Example: C(Ti, v, Tj) = 0(102) corresponds to the sequence: Ti does not store, Ti stores in cell 1, Ti does not store, Ti store in cell 2, Ti stores in 1, etc.

SchedMCore is able to compute whether the set of tasks is schedulable. SchedMCore Runner provides a multi-threaded (POSIX) simulation in shared memory. It can be seen as a user-level scheduler for RTOS or Linux.

8.2.2 Specific Hardware Improved for Off-line Mapping

Manel Djemal presents the Programmable DSPIN architecture [7]. It is a NoC architecture especially designed to improve mapping of hard real-time applications on many-core architectures. This architecture is designed to implement both optimal computation and communication.

The main contribution is to provide a NoC whose fair routers are replaced by programmable routers. The routers statically route the packets according to a micro-program. The idea is that a fair policy as a round robin between several input sources is not always desirable for predictability. Static routing allows to precisely optimize for a specific application.

To the best of our knowledge, this architecture only exists in simulation.

— 9 —

Conclusion and future work

The input of our problem is a synchronous program (multi-periodic or not) and the information on the activation periods of the nodes. If we consider a many-core architecture whose transmission times and computation times are bounded, and the core timers are synchronized (mesosynchronous) then we provide a deterministic semantics for the communication between the nodes of the synchronous program. We give the implementation of this communication semantics for nodes communicating through the NoC. Finally, we implement a full multi-periodic synchronous program on the Kalray MPPA.

Running synchronous programs on a many-core architecture is an important problem that leads to a lot of sub-problems. Hence, for now, industrial applications are not yet possible. Our work solves a required part of this problem which is the mapping inter-cluster communication. Nevertheless, we provide a proof of concept showing that mapping a reactive program on a many-core architecture is possible.

There are several directions of progress. The obvious first one is an improvement of our solution to use more than one core per cluster. Adding a level of mapping seems essential to fully take advantage of the many-core architecture. For the long term improvements, the aim is to industrialize the solution. There are several possibilities: either we make the distribution on the architecture fully automatic, or we modify a synchronous language (or create a library) to allow the developer to take advantage of the architecture. In both cases, we need to answer the questions: how to split the synchronous programs into parts to map on the architecture? For instance by node, group of nodes, group of nodes of same period, etc. And, how to map these parts on the architecture? For instance, one node per core, one group of nodes per cluster, etc.

In any case, if our solution targets the safety-critical systems in production, all the tools used during the implementation process must be certified.

— A —

Appendix: Reference

```
-- THigh
node Node1(init: bool) returns (x:int)
let
    x = if init then 0 else (0->pre x) + 1;
tel

-- THigh
node Node3(init:bool; x:int) returns (x2: int) -- Writer
let
    x2 = if init then 0 else 2*x;
tel

-- TLow
node Node2(init: bool) returns (y:int)
let
    y = if init then 0 else (0->pre y) + 2;
tel

-- TLow
node Node4(init: bool; i, y: int) returns (res: int) -- Reader
let
    res = if init then 0 else 3*y + i;
tel

-- "current" with an initialization value
node generic_current<<type ty>>(c: bool; v: ty; (x: ty) when c)
    returns (o:  ty);
var
    oncec: bool;
let
    oncec = c or (false -> pre oncec);
    o = if not oncec then v else current(x);
tel

node generic_UnitDelay<<type ty>>
    (clockA,clockB:bool; (input:ty) when clockA; val:ty)
returns ((output:ty) when clockB);
var
    currOutput: ty;
let
    currOutput = generic_current<<ty>>
        (clockA, val,(val when clockA) -> pre input);
    output = currOutput when clockB;
tel
```

```
node UnitDelay = generic_UnitDelay<<int>>;


node Connection(useless: bool) returns (currX, currY, currRes: int)
var
    seq: int;
    init, highC, lowC: bool;
    (N1, N3: int) when highC;
    (N2, N4: int) when lowC;
    x: int when highC;
    (y, z, res: int) when lowC;
let
    init = true -> false;

    -- Define clocks
    seq = 0 -> ((pre seq + 1) mod 6);

    highC = (seq mod 2 = 0);
    lowC =  (seq mod 3 = 0);

    N1 = Node1(init when highC);
    N2 = Node2(init when lowC);
    N3 = Node3(init when highC, x);
    N4 = Node4(init when lowC, z, y);

    x = UnitDelay(highC, highC, N1, 0);
    y = UnitDelay(lowC, lowC, N2, 0);
    z = UnitDelay(highC, lowC, N3, 0);
    res = UnitDelay(lowC, lowC, N4, 0);

    currX = generic_current<<int>>(highC, -1, x);
    currY = generic_current<<int>>(lowC, -1, y);
    currRes = generic_current<<int>>(lowC, -1, res);
tel
```

— B —

Appendix: Implementation

```
/*
 * Global
 */
#include <mppa.h>
#include <mppaipc.h>
#include <mppa/osconfig.h>
#include <sys/time.h>

#define PATT_ACTIVE(n) (1&(pattern>>(PATTERN_PERIOD+PATTERN_INIT - 1 - n)))
#define MASK_21BITS_LSB 0x1FFFFF
#define IFCE 0

#define TLowClock  3*40000
#define THighClock 2*40000

// ----------------------------------------------
/*
 *   Reader.c (Cluster 4)
 */
// Pattern: 0(1)
#define PATTERNin_INIT 1

// Pattern: 0(1)
#define PATTERN_INIT 1

int main(int argc, char *argv[])
{
    initTimer();

    // Input z
#define TAGz 7
    uint64_t recv_2X, pre2X = 0;
    __k1_dnoc_configure_rx(TAGz, _K1_NOCV2_INCR_DATA_NOTIF,
        &recv_2X, sizeof(recv_2X), 0);
    __k1_dnoc_activate_rx_ext(IFCE, TAGz);

    // Input y
#define TAGy 13
    uint32_t recv_y = 0, y = 0;
    __k1_dnoc_configure_rx(TAGy, _K1_NOCV2_INCR_DATA_NOTIF,
        &recv_y, sizeof(recv_y), 0);
    __k1_dnoc_activate_rx_ext(IFCE, TAGy);

#define TAGres 1
#define TAGdst_res 12
```

```
#define ROUTEres 0x80800003     // From 8 to 10
#define FIRST_DIRECTIONres 0x2
    union __device_channel_header_t headerres;
    headerres._.route = (uint64_t) (MASK_21BITS_LSB & ROUTEres);
    headerres._.multicast = 0;
    headerres._.tag = TAGdst_res;
    headerres._.eot = 1;
    headerres._.extended = 1;   // enable route extension
    headerres._.routex = (uint64_t) (ROUTEres >> 21);
    headerres._.protocol = 0;   // protocol: bit extension
    headerres._.valid = 1;
    __k1_dnoc_configure_tx(IFCE, TAGres, FIRST_DIRECTIONres,
        (uint64_t) headerres.dword, 0);

    uint64 end = readTimestamp();
    uint32_t initialization = 0;

    while (1) {

        // READ INPUTS
        if (initialization >= PATTERN_INIT) {
            // Read packet
            while (!__k1_dnoc_get_notification_counter_rx(TAGz))
                ;

            pre2X = __k1_umem_read64(&recv_2X);
            __k1_dnoc_event_cntr_load_and_clear(TAGz);
        }
        // Pattern 0(1)
        if (initialization >= PATTERNin_INIT) {
            // Read y (same speed)
            while (!__k1_dnoc_get_notification_counter_rx(TAGy))
                ;
            y = __k1_umem_read32(&recv_y);
            __k1_dnoc_event_cntr_load_and_clear(TAGy);
        }

        // COMPUTATION
        uint64_t res = 3*y + pre2X;

        // Pattern (1)
        __k1_dnoc_send_data(IFCE, TAGres, &res, sizeof(res), SEND_EOT);

        // Pattern
        if (initialization < PATTERN_INIT) {
            initialization++;
        }

        // Period
        end = end + TLowClock;
        assert(readTimestamp() <= end);
        us_loop_until(end);
    }
}
// -----------------------------------------------
```

```c
/*
 * Writer.c (Cluster 3)
 */
// Pattern: 0(1)
#define PATTERNin_INIT 1

// Pattern: 10(110)
#define PATTERN_INIT 2
#define PATTERN_PERIOD 3
uint64_t pattern = 0b10110;

int main(int argc, char *argv[])
{
    initTimer();

#define TAGz 3
#define TAGdst_z 7
#define ROUTEz 0x80400037       // From 1 to 8
#define FIRST_DIRECTIONz 0x1
    union __device_channel_header_t headerz;
    headerz._.route = (uint64_t) (MASK_21BITS_LSB & ROUTEqueue);
    headerz._.multicast = 0;
    headerz._.tag = TAGdst_queue;
    headerz._.eot = 1;
    headerz._.extended = 1;     // enable route extension
    headerz._.routex = (uint64_t) (ROUTEqueue >> 21);
    headerz._.protocol = 0;     // protocol: bit extension
    headerz._.valid = 1;
    __k1_dnoc_configure_tx(IFCE, TAGz, FIRST_DIRECTIONqueue,
                    (uint64_t) headerz.dword, 0);

    // Initialization of the incoming channel
#define TAGx 12
    int64_t recv_i, i;
    __k1_dnoc_configure_rx(TAGx, _K1_NOCV2_INCR_DATA_NOTIF,
        &recv_i, sizeof(recv_i), 0);
    __k1_dnoc_activate_rx_ext(IFCE, TAGx);

    uint64_t end = readTimestamp();
    uint32_t period = 0, periodIn = 0;
    srand(17);
    while (1) {
        // READER INPUTS
        // Pattern 0(1)
        if (periodIn >= PATTERNin_INIT) {
            // Read x
            while (!__k1_dnoc_get_notification_counter_rx(TAGx))
                ;
            i = __k1_umem_read64(&recv_i);
            __k1_dnoc_event_cntr_load_and_clear(TAGx);
        } else {
            periodIn++;
        }
```

```
        // COMPUTATION
        int64_t o = 2*i;

        // WRITE OUTPUTS
        if (PATT_ACTIVE(period)) {
            __k1_dnoc_send_data(IFCE, TAGz, &o, sizeof(o), SEND_EOT);
        }
        // Pattern
        period++;
        if(period == PATTERN_INIT + PATTERN_PERIOD) {
            period = PATTERN_INIT;
        }

        end = end + THighClock;
        assert(readTimestamp() <= end);
        us_loop_until(end);
    }
}
```

[1] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete. Cycle-static dataflow. *Signal Processing, IEEE Transactions on*, 44(2):397–408, Feb 1996.

[2] Paul Caspi and Oded Maler. From control loops to real-time programs. In *Handbook of networked and embedded control systems*, pages 395–418. Springer, 2005.

[3] Paul Caspi, Norman Scaife, Christos Sofronis, and Stavros Tripakis. Semantics-preserving multitask implementation of synchronous programs. *ACM Transactions in Embedded Computing Systems*, 7:1–40, 2008.

[4] Albert Cohen, Marc Duranton, Christine Eisenbeis, Claire Pagetti, Florence Plateau, and Marc Pouzet. N-Synchronous Kahn Networks: a Relaxed Model of Synchrony for Real-Time Systems. In *ACM International Conference on Principles of Programming Languages (POPL 06)*, Charleston, South Carolina, USA, January 2006.

[5] Mikel Cordovilla Mesonero. *Environnement de développement d'applications multipériodiques sur plateforme multicoeur. : La boîte à outils SchedMCore*. PhD thesis, 2012. Thèse de doctorat dirigée par Boniol, Frédéric et Pagetti, Claire Sûreté de logiciel et calcul de haute-performance Toulouse, ISAE 2012.

[6] B.D. de Dinechin, D. van Amstel, M. Poulhies, and G. Lager. Time-critical Computing on a Single-chip Massively Parallel Processor. In *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014*, pages 1–6, March 2014.

[7] Manel Fakhfakh (Djemal). *Réconcilier performance et prédictibilité sur un many-coeur en utilisant des techniques d'ordonnancement hors-ligne*. PhD thesis, 2014. Thèse de doctorat dirigée par Munier-Kordon, Alix Informatique Paris 6 2014.

[8] Christian Ferdinand, Florian Martin, and Reinhard Wilhelm. Applying compiler techniques to cache behavior prediction. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time System (LCTRTS) Las Vegas, Nevada*, pages 37–46, 1997.

[9] Hanan Kanso. Implementing critical systems on many-core architectures: Towards solutions preserving determinism. Master's thesis, Grenoble INP, Ensimag, 2014.

[10] Daniel Pilaud, N Halbwachs, and JA Plaice. Lustre: A declarative language for programming synchronous systems. In *Proceedings of the 14th Annual ACM Symposium on Principles of Programming Languages (14th POPL 1987). ACM, New York, NY*, volume 178, page 188, 1987.

[11] Kalray S.A. *MPPA®ACCESSCORE - POSIX Programming Reference Manual*, 2014.

[12] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7(3):36:1–36:53, May 2008.

[13] E. Yip, P.S. Roop, M. Biglari-Abhari, and A. Girault. Programming and timing analysis of parallel programs on multicores. In *International Conference on Application of Concurrency to System Design, ACSD'13*, pages 167–176, Barcelona, Spain, July 2013. IEEE.