A Compilation Flow for Parametric Dataflow: Programming Model, Scheduling, and Application to Heterogeneous MPSoC

Mickaël Dardaillon, Kevin Marquet, Tanguy Risset, Jérôme Martin, Henri-Pierre Charles

A Compilation Flow for Parametric Dataflow: Programming Model, Scheduling, and Application to Heterogeneous MPSoC

Mickaël Dardaillon * Kevin Marquet Tanguy Risset
Université de Lyon, Inria,
INSA-Lyon, CITI-Inria,
F-69621 Villeurbanne, France
firstname.lastname@insa-lyon.fr

Jérôme Martin
Univ. Grenoble Alpes,
F-38000 Grenoble, France
CEA, LETI, Minatec campus,
F-38054 Grenoble, France
jerome.martin@cea.fr

Henri-Pierre Charles
Univ. Grenoble Alpes,
F-38000 Grenoble, France
CEA, LIST, Minatec campus,
F-38054 Grenoble, France
henri-pierre.charles@cea.fr

Abstract

Efficient programming of signal processing applications on embedded systems is a complex problem. High level models such as Synchronous dataflow (SDF) have been privileged candidates for dealing with this complexity. These models permit to express inherent application parallelism, as well as analysis for both verification and optimization. Parametric dataflow models aim at providing sufficient dynamicity to model new applications, while at the same time maintaining the high level of analyzability needed for efficient real life implementations.

This paper presents a new compilation flow that targets parametric dataflows. Built on the LLVM compiler infrastructure, it offers an *actor based* C++ programming model to describe parametric graphs, a compilation front-end providing graph analysis features, and a retargetable back-end to map the application on real hardware. This paper gives an overview of this flow, with a specific focus on scheduling. The crucial gap between dataflow models and real hardware on which actor firing is not atomic, as well as the consequences on FIFOs sizing and execution pipelining are taken into account.The experimental results illustrate our compilation flow applied to compilation of 3GPP LTE-Advanced demodulation on a heterogeneous MPSoC with distributed scheduling features. This achieves performances similar to time-consuming hand made optimizations.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—Data-flow languages; D.3.3 [*Programming Languages*]: Processors—Retargetable compilers

Keywords data flow, programming model, heterogeneous MPSoC, compiler, scheduling

* This work is sponsored by Région Rhône Alpes ADR 11 01302401.

1. Introduction

Implementation of signal processing algorithms in the dataflow programming model is an active research area, and many popular signal processing environments (Simulink, Labview, etc.) already use this paradigm. Dataflow programming models are natural candidates for streaming applications. They allow both static analysis and explicit parallelism, making them suitable for embedded applications such as packet processing, cryptography, telecommunications, video decoding, etc. This is of particular interest in the wireless digital telecommunication domain where implementation of wireless protocol has to be computationally efficient and predictable for real time communication, but also energy efficient to be embedded in mobile phones.

"Advanced" 4G (e.g. LTE-Advanced) and forthcoming 5G wireless protocols, the advent of SDR technologies and the development of cognitive radio networks reveal new challenges for expressing and compiling wireless applications. The physical layer of wireless protocols have a dynamic behavior and require fast dynamic reconfigurations which are not possible with today's wireless devices. These technological trends have re-activated past research axes such as dynamic dataflow compiling or distributed signal processing algorithm. In particular, the need for flexible but still verifiable programs has led very recently to the appearance of new parametric dataflow Models of Computation (MoC).

A typical example to illustrate dataflow dynamicity comes from LTE-Advanced: the type of modulation used to decode samples in a LTE-Advanced frame (i.e. QPSK, 16-QAM, etc.) is indicated within the frame itself. Hence the hardware should be able to adapt to this modulation within a few micro-seconds. Classical dataflow programming models that cannot express dynamic behavior need to be extended [7].

LTE-Advanced decoding, as well as 5G telecommunications protocols, will run on dedicated system on chip (SoC) with sufficient processing power (order of 40 GOPS for LTE-Advanced [12, 31]) and reasonable power consumption (less than 500 mW). The challenge with these SoCs is to set up a real compilation flow that takes advantage of the hardware acceleration while retaining portability. There already exist some implementations of LTE-Advanced being commercially deployed. Unfortunately these implementations are highly dedicated to a single architecture and are usually manually tuned to meet the hard performance and power-efficiency constraints. Our proposal is a step towards a more generic approach: compiling SDR waveforms from high level dataflow representations rather than manual tuning.

The contributions provided by this work are:

1. A new high level format for expressing parametric dataflow graphs based on semi-static actors. This format permits to express parametrized dataflow graphs in an high level programming model so as to describe, for instance, a MIMO receiver with N antennas (detailed hereafter). As N should be known before compilation we refer to this format as semi-static.
2. A new static analysis paradigm called *micro-schedule* that permits a more precise analysis of deadlock when mapping parametrized data-flow graph to real architecture. We have used model checking to show the applicability of our micro-schedule paradigm to the specific problem of advanced actor pipelining in the context of dedicated target architecture with small buffers.
3. The first complete compilation flow for the Magali platform [11]. Magali programs are usually tuned by hand. Our compilation framework has been instantiated for the Magali architecture (with part of it still being manual, such as mapping for instance) and we obtain performance of the compiled programs which are roughly equivalent to manually tuned programs. This highlights the fact that our generic compilation framework can adapt to very dedicated architectures such as Magali.

The paper is organized as follows: section 2 present our compilation framework, sections 3 and 4 present parametric synchonous dataflow and the micro-schedule proposal. Experimental results are presented in section 5 and related work are presented in section 6.

2. Overview of Compilation Framework

In this section, we describe simultaneously the input format used to express parametric dataflow graphs and the compilation flow that we have built to compile these programs. A dataflow compilation framework should compile high level specifications of a dataflow representation (we use the parametric dataflow paradigm) and produce the executable codes needed to program the target platform. To be retargetable, it should also take as input a description of the target architecture. As the target architecture often contains dedicated processors, we use the term "hardware IP" (or simply IP for *intellectual property*) to refer the various kind of processors (general or dedicated) that are present in the target machine.

Our dataflow compilation framework, illustrated in Fig. 1, is split into two phases: i) *Front-End* for parsing and analysis of the dataflow graph, and ii) *Back-End* for mapping, scheduling and assembly code generation. Only the back-end uses the target architecture characteristics as input.

The analysis checks that the data flow graph is correct. The mapping phase associate nodes of the data flow graph with hardware IPs on which they will be executed. The scheduling schedules, o n each IP, the order of execution of the different nodes assigned to it. Then, the code is generated for each IP. Depending on the target architecture, a central controller is used for global synchronization or the synchronization can be distributed.

Our compilation also includes features that are not present in other dataflow compilation frameworks. These features, detailed further in the paper are: handling of parametric dataflow programs, complex dataflow graph construction, buffer size checking on scheduled application, and code generation for complex heterogeneous SoCs.

2.1 Compiler Front-End

The compiler front-end is in charge of lowering the high-level representation to intermediate representation (IR). It is based on the LLVM framework [3]. We present now our actor based programming model and how it is processed down to IR with the added graph representation.

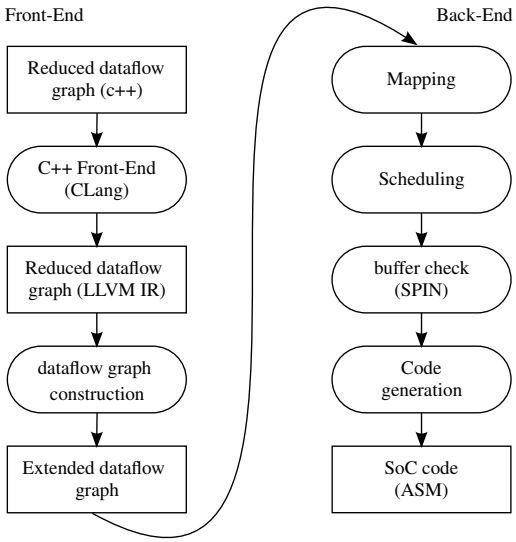


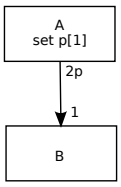
Figure 1. Proposed compilation flow from dataflow to heterogeneous SoC.

2.1.1 Parametric Dataflow Format

The input format follows the *Schedulable Parametric DataFlow* (SPDF) model of computation propose by Fradet et al. in [16]. In a dataflow graph (DFG), from the classical dataflow model of computation, the integers on the arcs represent the number of samples produced or consumed by the actor at each execution. In SPDF, this number can be a symbolic parameter whose value is produced by an actor. A simple SPDF graph is represented on Fig. 2b, the `set p[1]` indicates that the actor `A` produces a new value for parameter p each time it fires. Also, the graph indicates that actor `A` defines p and then output $2p$ data.

```
//actors declaration
class A : public Actor {
  public:
    PortOut<int> out;
    ParamOut p;
    void compute() {
      [...]
      p.set([...])
      for (i=0; i<p; i++)
        out.push([...]);
    }
[...]
//Graph declaration
A actorA;
B actorB;
actorB.in <= actorA.out;
[...]
```

(a) Simple SPDF graph in PaDaF



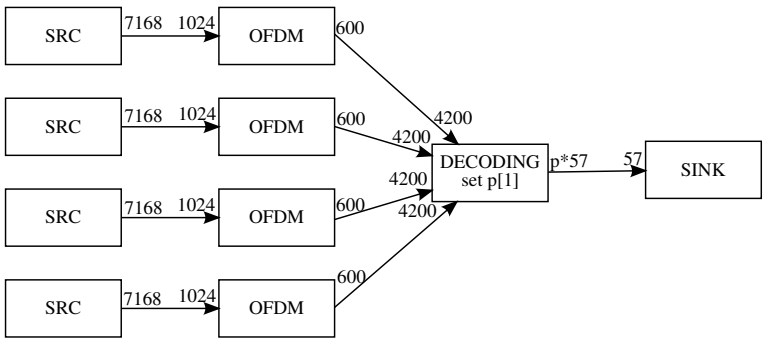
(b) simple SPDF graph

Figure 2. Simple SPDF graph expressed in PaDaF

The input format we propose, called PaDaF (*Parametric Dataflow Format*) is an actor programming model based on c++. It consists of a set of classes allowing to describe a parametric data flow graph. Fig. 2a present parts of the program implementing the simple SPDF graph of Fig. 2b, it illustrates how actors are declared and shows specific classes for data ports (`PortOut` class) and for parameters produced (`ParamOut` class). Our format is close to systemC [28]. The originality of PaDaF is that it permits to describe the DFG in a *closed form* (or *reduced graph*) using classical control structure

```
[...]
SRC src[NB_ANT];
OFDM fft[NB_ANT];
DECODING mimo(NB_ANT);
SINK sink;
for(i = 0; i < NB_ANT; i++) {
  fft[i].in <= src[i].out;
  mimo.in[i] <= fft[i].out;
}
sink.in <= mimo.out;
[...]
```

(a) program excerpt (Reduced graph).



(b) Corresponding Extended graph (for 4 antennas): SPDF model

Figure 3. PaDaF simple input program (Fig. 3a) representing a MIMO receiver with `NB_ANT` antenna. The graph on the right (Fig. 3b) has been extended for `NB_ANT=4` and corresponds to a parametric data-flow graph in the SPDF model, p is a parameter instantiated at runtime: the number of data sample decoded in the frame by the `DECODING` IP is computed after receiving some of the 4200 samples of the frame itself.

instructions (`for` loop for instance) and any C++ structure for that matter. The only constraint is that the graph structure is static, i.e. all the information to *draw* the graph has to be known at compilation time. Fig. 3a illustrates the use of PaDaF to describe, in a closed form, a SPDF graph with a parametric number of `OFDM` nodes (`NB_ANT` is the name of this parameter). The *extended* data flow graph of this PaDaF program is shown on Fig. 3b for `NB_ANT=4`.

Each actor has a single `compute` method that represents the execution of one iteration of the actor. The code of this method is written in C++ and uses various `push`/`pop` intrinsics to send/receive data and parameters. An excerpt of the `compute` method of the `DECODING` actor is shown on Fig. 4

```
void DECODING::compute() {
  [...]
  for(i = 0; i < NB_ANT; i++) {
    val[i] = in[i].pop();
  }
  [...]
  p.set(size);
  [...]
```

Figure 4. PaDaF program excerpt for the core computation.

Choosing C/C++ language for the core code of the actors exhibits many advantages: it allows designers to reuse legacy code and highly optimized tools such as C compilers; it does not require to learn a new language; and it permits easy simulation and functional validation. Moreover, the support of a general purpose language for describing the graph *structure* greatly simplifies the specification of some applications as it provides important capabilities such as the ability to iterate for the construction of complex structures (channels of the MIMO receiver illustrated on Fig. 3). As a matter of fact, arbitrary C++ code can be used for the DFG construction, as long as it produces a static graph at compilation time. This approach raises challenges that we solve in the following paragraph.

2.1.2 Dataflow Graph Construction

From the program describing the reduced DFG (e.g. Fig 3a), we build the extended DFG (e.g. Fig. 3b). First of all, we compile the C++ code using the Clang compiler [1]. We obtain a low-level description of the graph in the LLVM IR bytecode that we call the *reduced graph representation*. In this representation, the multiple instances of the same actor are not instantiated and loop structures are kept. Building the *extended graph representation* in which the different instances exist is difficult because the reduced graph is described using arbitrary complex control structures. We use here a technique already applied to SystemC [27]: we *execute* the program to create all instances.

At this point, the DFG lives in memory, we can work with the internal memory representation as a classic C++ object: to generate the final code, we need to process the `compute` method of each actor. This method describes the actor behavior during a cycle, it contains input/output instructions operating on ports (`pop`/`push`). For instance, in the instruction `in[i].pop()` on Fig. 4, the object `in[i]` represents a port. In general it can be expressed by arbitrarily complex code, e.g. an array accessed within a loop in Fig. 4. We need to expand this code and to link each port access with its corresponding port in the extended DFG to continue the compilation flow.

We rely on the solution used in [27], divided in three parts:

1. We use a slicing algorithm on the LLVM IR to extract *only* the code that is used to build input/ouptut instructions such as `in[i].pop()`. If these instructions are dependent of the loop index, the loops are unrolled (at this point, the loop range has to be known).
2. All these instructions are put in a new LLVM function.
3. This function is executed by an instance of the LLVM Just-In-Time compiler. The result is the address of `in[i]` (for each `i`), which points to an actor port of the extended dataflow graph in memory.

After that, the execution is interrupted and a callback function is executed to launch the Back-end phase of our compiler using the extended graph present in memory as internal representation.

2.2 Compiler Back-End

Once we have built our graph, the Back-End of the compiler is in charge of specializing the program for the platform targeted. We describe the different steps of this specialization.

Mapping Mapping actors on hardware cores on the basis of an Architecture Description Language has been the focus of numerous past research works [9, 10, 22, 24]. However they did not provided a satisfactory solution. We did not investigate in this direction. Given that the granularity of the actors in the SDR domain is quite large (an actor can contain a full FFT), we assume that this mapping is done manually, as it is already the case in many existing heterogeneous SoC programming environments. Hence, in our flow,

the hardware core on which each actor executes is given by the programmer.

Scheduling Once the mapping is performed, the compiler computes a schedule for each core. The simplest schedule is to run all actors concurrently on a core and postpone the scheduling to runtime by data synchronization. However, dedicated platforms such as Magali [11] does not support runtime scheduling. In such case, we generate a static schedule for the execution of the different actors on the core. More details on the scheduling methodology can be found in section 3.

Buffer Checking Given the constraints introduced by the application specialization to a given platform (e.g. static scheduling, memory constraints), we introduce a buffer size verification step before the code generation. This step generates a model of the application's communication on the targeted platform. The model is generated in the PROMELA language, and is run on the SPIN model checker [4] (explained in section 4).

Each core is represented by a concurrent process, containing the static schedule of the actors mapped on the core. Each actor is reduced to its communications on the graph. All the communications are represented as access to blocking FIFOs. This model permits to control the absence of deadlock due to memory constraints. The verification process and the communication representation as the *micro-schedule* are explained in section 4. Evaluation of the verification step on several applications is presented section 5.3.

Code Generation Our code generation is original in two ways. First, it is able to generate communications from high-level DFG representation, while taking advantage of the platform-specific mechanisms. Second, it is able to generate distributed scheduling and synchronization based on the extended DFG representation. Depending on the platform, it gives the ability to have completely distributed control, or to have a centralized controller scheduling the different cores.

For example, on the Magali platform presented in section 5.1, parameter synchronization has to be done by the central CPU. In this case, each core is associated with a thread on the central CPU managing the parameter. The remaining schedule is managed locally by the core. This approach differs from classic telecommunication control, where applications are split into different *phases*, each one running a static data flow, whereas *phase* transitions reflect parameter changes. By relaxing the control constraints, we aim to take advantage of the potential pipelining introduced by the data flow MoC. Evaluation of our compiler in terms of development time and generated code performance for the Magali platform is done in section 5.3.

3. Parametric Dataflow Scheduling

Scheduling is a key optimisation problem for efficient map of dataflow applications on a real hardware. In this section, we show how the well-known case of static dataflows scheduling has been recently extended to parametric dataflows, bringing more flexibility to model and map real life applications. We also show some limitations of these approaches when targeting certain kinds of real hardware platforms.

3.1 Scheduling Static Dataflows

Dataflow languages rely on a MoC in which a program is usually formalized as a directed graph $\mathcal{G} = (\mathcal{A}, \mathcal{E})$. An *actor* $v \in \mathcal{A}$ represents a computational module or a hierarchically nested subgraph. A directed edge $e = (X_1, X_2) \in \mathcal{E}$ represents a FIFO buffer from its source actor X_1 to its destination actor X_2. The execution (or firing) of an actor X consumes data tokens from its incoming edges and produces data tokens on its outgoing edges. The number of tokens produced on an outgoing edge or consumed on an incoming edge by an actor at each firing is called a *rate*. It is usually represented as a label on the edges ends. In the following, incoming and outgoing edges are also called input and output edges, respectively.

Dataflow graphs follow a data-driven execution: an actor can be fired only when enough data samples are available on its input edges. From the model point of view, firing of actor X is an atomic operation, which consumes from each of X's input edge the amount of samples corresponding to the edge's rate, and produces on each of X's output edge the number of data sample given by the edge's rate.

Many dataflow-compliant programming models have been proposed for specific applications. An important category comprises dataflows where the graph topology and rates is static, i.e. fixed and known at compile-time. A famous example of such static dataflow representation is called *Synchronous DataFlow* (SDF [25]). A major advantage of SDF is that, if it exists, a bounded schedule can be found statically. Such a schedule ensures that each actor is eventually fired (ensuring *liveness*) and that the graph returns to its initial state after a certain sequence of firings (ensuring *boundedness* of the FIFOs). A sequence that verify these properties with the minimum number of firing of each actor is called an *iteration*. This minimum number of firing can be obtained by solving the so-called system of balance equations. This system is made of one equation per edge $e = (X_1, X_2)$ of the form

$$\#X_1 \cdot r_{e,1} = \#X_2 \cdot r_{e,2} \tag{1}$$

where $\#X_1$ and $\#X_2$ denote the number of firings of the actors X_1 and X_2 in an iteration, $r_{e,1}$ is the output rate of X_1 on edge e, and $r_{e,2}$ is the input rate of X_2 on edge e. A graph is *consistent* if its system of balance equations has non-null solutions. The minimal solution of the balance equations is called *repetition* vector (or *iteration* vector) [25].

3.2 Scheduling Parametric Dataflows

Many other MoCs have been proposed to relax the condition that the number of tokens should be known at compile time. These related works are detailed in section 6. Among them, SPDF has shown interesting properties, being used to program homogeneous multi-core architectures [5] as well as heterogeneous systems-on-chip [15].

SPDF [16] is a dataflow MoC where the number of tokens can be parametric. Parameters are represented by a set of symbolic variables p,q,... which can take only integer values. In SPDF, input and output rates can be integers, parameters, or products of these two. The reader should refer to [16] for a more formal definition of SPDF.

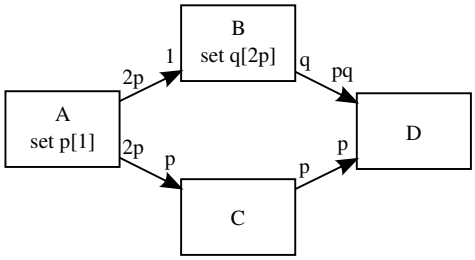


Figure 5. Example of dataflow application in the SPDF model of computation with repetition vector $AB^{2p}C^2D^2$.

Fig. 5 shows an example of SPDF graph with four actors and two parameters: p and q, the notation $q[2p]$ in actor B indicates the *change period* of the parameter: q is set every $2p$ execution of B. In

this example, the iteration vector of the graph is: (A, B^{2p}, C^2, D^2), it is usually written in the following way: $AB^{2p}C^2D^2$ although it does imply a sequential ordering of the firings. A way to compute this vector is presented in [16].

A parameter cannot change anywhere in during the execution of the iteration. Allowing arbitrary parameter change period greatly complicates analysis of SPDF graphs, and of course not all parameter change period are valid. In this paper we choose, as it was done in other works following SPDF [5, 6], to impose that the parameters change only once per iteration, in practice at the beginning of the iteration.

Using this notation (e.g. $AB^{2p}C^2D^2$) for the iteration vector does not indicate when and where parameters are set and used. Fradet et al. [16] introduce the notion of *quasi-static schedule* which is a set of elements executed in a sequential manner. These elements are of three kinds:

- Executing n times the actor X. It is denoted X^n, where n can be a parametric expression.
- Actor X getting the value of a parameter p is denoted $get_X(p)$ (or $get(p)$ when it is not ambiguous),
- Actor X setting the value of a parameter p, denoted $set_X(p)$.

The setting of a parameter by an actor is performed after actor firing and the getting of a parameter is performed before actor firing. A quasi-static iteration vector is therefore a repetition of quasi-static schedules of each actor possibly interleaved with production and consumption of parameters. Because of our assumption concerning parameter change period, it is safe to impose that each parameter consumption is performed before the execution of the actor and each parameter production is performed after the execution of the actor. For instance, for the graph of Fig. 5, the quasi-static schedule of the graph corresponding to the extension of the iteration vector with parameter synchronization, is:

$$\begin{array}{l}(A; set_A(p)) \quad (get_B(p); B^{2p}; set_B(q)) \\ \qquad (get_C(p); C^2) \quad (get_D(p); get_D(q); D^2)\ .\end{array} \tag{2}$$

If the SPDF graph is to be executed on a single computing resource, one can define a sequential schedule of the iteration. This sequential schedule is obtained by a topological sort of the graph if it is acyclic. This method can be extended to cyclic graph on certain conditions [8]. Finding a sequential quasi-static schedule for a SPDF graph is beyond the scope of this paper. Based on previous works [16], we assume that we have a valid sequential schedule. For instance, this is a valid sequential schedule obtained by topological sort of the SPDF graph of Fig. 5:

$$\begin{array}{l}(A; set_A(p); get_C(p); C^2; get_B(p); B^{2p}; set_B(q); \\ \qquad\qquad get_D(p); get_D(q); D^2)\ .\end{array} \tag{3}$$

Such a global sequential schedule of our SPDF graph can be easily used as a starting point for graph scheduling onto a multi-core platform. In the following, we use the term IP to show that we make no particular assumption on the nature of processing elements of the hardware platform (general purpose processors, DSPs, dedicated hardware). In the general case, one or several graph actors may be mapped on a given IP. Corresponding schedule can be easily built by simply scheduling each mapped actor *in the order it was scheduled in the global sequential schedule*.

Consider for instance, the simple SPDF graph of Fig. 5 executed on two IPs: IP_1 and IP_2. If A and C are mapped on IP_1 and B and D on IP_2, we obtain a valid multi-core schedule by scheduling on each core, the actors in the order it was in the sequential schedule:

$$\begin{array}{l}S_{IP_1} = (A; set_A(p); get_C(p); C^2) \\ S_{IP_2} = (get_B(p); B^{2p}; set_B(B)\ q; get_D(p); get_D(q); D^2)\ .\end{array} \tag{4}$$

If parameters are shared by actors mapped the same IP, we can remove redundant synchronization. We then obtain the following schedule:

$$\begin{array}{l}S_{IP_1} = (A; set(p); C^2) \\ S_{IP_2} = (get(p); B^{2p}; set(q); D^2)\ .\end{array} \tag{5}$$

3.3 Limitations with Existing Approach

In many works dealing with classical SDF schedule [8, 17, 25], a specific focus is made on minimizing the size of the FIFOs needed to forbid deadlock. Indeed, FIFO size optimization is often a major concern in real life implementation, where available memory is often limited, because of its cost and power consumption. In embedded hardware platforms for example, memory reserved for data communications between actors is usually very restricted. for instance the Magali platform only allows 64 bytes of data in its fixed-size communication FIFO.

However, classical approaches with dataflow formalism are often ill-suited to correctly model data transfers between actors on a real platform. Consider example of Fig. 5, and suppose the quasi-static schedule (5) from previous section. With this scheduling formalism, we need a FIFO of size $|AB| = 2p_{max}$ between A and B (usually a maximal value p_{max} for each parameter is specified allowing to assess bound for the FIFOs). However, B could be triggered as soon as one token is produced on the channel. Hence, if A is able to output data one token at a time and if the platform provides the necessary synchronization facilities (basically blocking write operation on FIFOs), the size of the required FIFO can be limited to one.

In practice, actor firing do not strictly follow the $read\ inputs \rightarrow compute \rightarrow write\ outputs$ model. Computation may start with only part of input data, and the first output data samples may be sent before all input samples are read. Size of FIFOs can therefore be optimized further if this behavior is taken into account.

In the particular case of SPDF, parameter synchronization between quasi-static schedules is another example of required model improvements: as is, the $set(p) \rightarrow get(p)$ dependency in schedule (5) forbids any firing of B before A has finished to produce all data samples. However, computation of parameter p may usually be done before the token production, i.e. the sequentiality $A; set_A(p)$ in the model is artificial and does not reflect real behavior. In next section, we introduce *micro-schedules* as a way to explicitly express the relative dependencies between production and consumption of data and parameters.

4. Micro-Schedules

In this section, we introduce our refinement to the quasi-static scheduling formalism: the *micro-schedule* formalism. Then we show how we have used micro-schedule to check in a more precise way, the consistency between the FIFO sizes of the actual target architecture and the schedule of the actors.

4.1 Refining Quasi-Static Schedules

The quasi-static schedule formalism was obtained by adding the production and consumption of parameters in the scheduling. We propose a second refinement which consists in adding the production and consumption of *each token*. This is what we name *micro-schedule*.

Micro-schedules express the sequential order of input and output operation of each actor. Note that this introduces constraints related to the target architecture: is this order fixed? is it statically known? can it rely on runtime decisions of the execution engine? In our study, we assume that the micro-schedule is quasi-static and known at compilation time. It is extracted from actor's computation code for processors, or predefined for hardware accelerators IPs.

Formally, the micro-schedule for a SPDF graph includes the following instructions in addition to the components of quasi-static schedules introduced in Section 3.2:

- Actor X sending n tokens to actor Y is denoted $push_{XY}(n)$
- Actor X receiving n tokens from actor Y is denoted $pop_{XY}(n)$
- Test for actor n^{th} execution during an iteration is denoted $i = n?$

Extra care is needed for parameter synchronization in this formalism. As we have seen in section 3.2, parameters are fixed for the whole iteration, meaning that, in the general case, parameter production and consumption is not done at each actor execution. We explain in the following paragraphs where parameter production and consumption should be indicated.

Parameter Production A Parameter is produced by an actor and should therefore be included in its micro-schedule and not in the micro-schedule of the IP on which it is mapped. The simplest case is an actor X which is fired only once per iteration: it produces a new parameter value at each execution, which makes the inclusion of the $set_X(p)$ inside the micro-schedule straightforward. If the actor is fired several time in an iteration, the $i = n?$ test operator is mandatory to precise which actor execution enables the production of the new parameter value. Usual cases are parameter production at the beginning or at the end of the iteration ($i = 1?$ or $i = \#X?$ respectively), but other cases are allowed.

Parameter Consumption Parameters consumption are provided in IP micro-schedule rather than in actor micro-schedule for two purposes. The first is when an actor is scheduled a parametric number of times (e.g. $(get(p); X^p)$). In this case, getting the parameter is done in the IP schedule unambiguously. The second possible use of a parameter is when an actor uses its value during its execution, e.g. produces or consumes a parametric number of tokens. In this case the parameter value is used inside the actor, and the $get(p)$ could be integrated in the micro-schedule. However, the refresh rate of the parameter is imposed by the scheduling of the whole graph. To respect this constraint, we keep the parameter's `get` outside the actor's micro-schedule, dissociating the synchronization imposed by the scheduling from the parameter usage inside the actor.

A valid micro-schedule for the actors of Fig. 5 could be:

$$\begin{array}{l} \mu S(A) = \big(set(p); \big(push_{AB}(1); push_{AC}(1)\big)^{2p}\big) \\ \mu S(B) = \big(pop_{AB}(1); i = 1? set(q); push_{BC}(q)\big) \\ \mu S(C) = \big(pop_{AC}(p); push_{CD}(p)\big) \\ \mu S(D) = \big(pop_{BD}(q); pop_{CD}(1)\big)^{p} \ . \end{array} \quad (6)$$

Again, this micro-schedule can be used for multi-core scheduling. Assuming a mapping of actors A and C on IP_1, B and D on IP_2, the multi-core schedule (5) is changed to reflect the setting of parameters inside the micro-schedules :

$$\begin{array}{l} S_{\text{IP}_1} = (\mu S(A); \mu S(C)^2) \\ S_{\text{IP}_2} = (get(p); \mu S(B)^{2p}; \mu S(D)^2) \ . \end{array} \quad (7)$$

In this example, we can reduce the FIFO to $|AB| = 1$, as well as $|CD| = p_{max}$. Building on this micro-schedule formalism, we solve the following problem in the next section: given a multi-core quasi-static micro-schedule of a SPDF graph on an architecture, are the FIFOs between the IPs of the architecture sufficiently large to avoid deadlock?

4.2 Checking Buffer Requirements

In previous section, we introduced the concept of micro-schedule to describe actors behavior in a dataflow graph. The execution of micro-schedule is guaranteed to be correct, provided that we have sufficiently large FIFOs. However, since on a real platform, the size of the buffers may be fixed and of small size, we now want to ensure that a given micro-schedule will execute correctly with available buffer sizes.

To a given micro-schedule μS correspond several *real* execution traces, because IPs run concurrently. Therefore, we want to check that, for *any* of these real executions, no deadlock is reached. Our approach is to walk through all possible execution traces thanks to the use of a model-checker.

Spin [4] is a model-checker targeting verification of multi-threaded software. It is extensively used in research and its maturity, efficiency and support for buffered message passing made it a good choice for our purpose. In addition, it has been used in previous work very similar [17] to our proposal. We now introduce our model using the Promela language with an example in the next section. Verification results on LTE-Advanced examples are presented in section 5.3.

4.2.1 Overview of the Promela Model

In order to prove the absence of deadlock, our Promela model is setup as follows:

- Each core is encoded as a Promela process `proctype`. As seen in section 4, we are refining the dataflow MoC by removing the hypothesis imposing atomic actor execution.
- The writing (resp. reading) of data is modeled by the `PRODUCE(c, n)` (resp. `CONSUME(c, n)`) primitives. This primitive models the writings (resp. reading) of `n` tokens in channel `c` by counting the number of tokens written (resp. read) in the channel. Hence the channel state is simply maintained by an integer indicating how many data are stored in it.
- Setting and getting parameters is modeled thanks to the `set()` and `get()` primitives. These primitives use `channel` type to model the transmission of parameters because we need the *values* or these parameters. Note that Spin is not a symbolic model-checker and that therefore, all possible values of the parameters are explicitly tested by Spin.

Based on this model, we ask SPIN to cover all possible executions, verifying the absence of deadlock. The next section illustrates the resolution with a simple example.

4.2.2 Micro-Schedule Modeling: Example

The example on Fig. 6 is the Promela code for checking buffer requirements of the execution of the graph on Fig. 2b with the following micro-schedules $\mu S(A) = \big(set(p); push_{AB}(2)^p\big)$ and $\mu S(B) = \big(pop_{AB}(1)\big)$. This execution takes place on an architecture with two IPs (`Core0` and `Core1`) and a FIFO of size $|AB| = 2$ between `Core0` and `Core1`. The IP schedules are $S_{Core0} = (\mu S(A))$ and $S_{Core1} = (get(p); \mu S(B)^{2p})$. The `select` statement is used to describe all possible values of the parameter p: $1 \leq p \leq 3$.

The `inline A()` and `B()` model the communications (i.e. micro-schedule) of actors A and B respectively. The actors A and B are mapped on `Core0` and `Core1`, the `proctype` processes implements the mapping using the multi-core schedule: `Core0` executes `A()`, `Core1` gets parameter `p` and executes `2p` times `B()`. The values of the `max` array indicate the maximum size of FIFOs in the architecture.

During the execution of the Promela code, the global integer array `ch` encodes the current number of tokens in the FIFOs, here only the FIFO between `Core0` and `Core1`. The macros `PRODUCE` and `CONSUME` model the production and consumption of tokens on the FIFOs. The execution of `PRODUCE(c,n)` (i.e. produce `n` tokens on FIFO `c`) checks the memory availability on FIFO `c` and add `n` tokens to the FIFO. The memory check is blocking, the process has to wait

```
#define PRODUCE(c,n) atomic{ch[c]+n <= max[c]
                        -> ch[c] = ch[c]+n;}
#define CONSUME(c,n) atomic{ch[c] >=n
                        -> ch[c] = ch[c]-n;}
#define SET(c,v) p[c]!v;
#define GET(c,v) p[c]?v;


int ch[1]; int max[1];      proctype Core1(){
chan p[1]=[1] of {int};       do :: {
                                int x, p_a;
inline A(){                     GET(0,p_a);
  int i, p_a;                   for(x:1..2*p_a) {
  select(p_a:1..3);               B();
  SET(0,p_a);                   }
  for(i:1..p_a) {             } od
    PRODUCE(0,2);           }
  }
}                           init{
inline B(){                   max[0] = 2;
  CONSUME(0,1);               atomic{
}                               run Core0();
                                run Core1();
proctype Core0(){             }
  do :: {                   }
    A();
  } od                      ltl deadlock {[]<> np_}
}
```

Figure 6. Promela code for the SPDF graph of Fig. 2b

until sufficient memory is available. This process is blocked if the FIFO is full, until sufficient memory is available in the FIFO to add the `n` tokens.

`CONSUME(c,n)` (i.e. consume `n` tokens on FIFO `c`) checks if sufficient tokens `n` are available on FIFO `c` before consuming them. `SET(c,v)` (set parameter `c` with value `v`) and `GET(c,v)` (get parameter `c` value in variable `v`) are used to set and get parameters. Note that we use parameter *channels* `chan p` to transmit parameters between actors because we need the *values* of the parameters in each actor. For `push` and `pop` modeling we only need the *numbers* of token exchanged, hence we use integers which reduce the complexity of the resolution.

Finally, the `ltl` (linear temporal logic) is used to set the correctness requirement. Our objective is to avoid deadlock due to communications, the `ltl` statement means: *always* (`[]`) *eventually* (`<>`) the system will be in a progress state (`np_`), that is to say: process is allways active. Using this statement, we challenge the model-checker to find one execution that will finish in a non progress state, i.e. a *deadlock*.

The execution of the program of Fig. 6 indicates that SPIN could not find a deadlock, meaning that FIFO deadlock will not occur on this architecture. The SPIN execution was almost immediate, using 328 states and 458 transitions; In section 5 we provide complexity measure of the Promela programs that we have use for LTE.

To conclude with section 3 and 4, we have shown that the association of micro-schedule with model checking offers, for the first time, a tool to control very precisely the pipeline between two IPs. In each architecture using hardware FIFOs, the sizes of these FIFO are small (because inter-IP hardware FIFOs are very costly), and these sizes are fixed of course. In next section we show how we have used this technique to check the absence of deadlock on the Magali platform, there was no such tool for that before.

5. Experimental Results

In this section we illustrate our compilation flow on the Magali chip. We start by presenting the experimental framework with the hardware platform and its simulation framework. Then we introduce our test applications based on LTE-Advanced, and finally we analyze the performance results.

5.1 Experimental Platform

Hardware Architecture The Magali chip [11], represented on Fig. 7a is a system on chip dedicated to physical layer processing of OFDMA radio protocols, with a special focus to 3GPP LTE-Advanced as reference application. It includes heterogeneous computation hardware, with very different degrees of programmability, from configurable blocks (e.g. FFT size and mask for OFDM modulation) to DSPs programmable in C. Main configuration and control of the chip is done by an ARM CPU, and communications between blocks use a 2D-mesh network on chip. Magali offers distributed control features, enabling to program sequences of computations at block level, thus limiting the required number of reconfigurations done by the CPU in the case of complex applications.

Simulation Framework The simulation framework is based on a SystemC *transaction level model* (TLM) of the Magali chip. Timing are extracted from the block synthesized in 65nm CMOS technology. The ARM central controller code is run on a QEMU virtual machine connected to the TLM model of the platform. Time synchronization between the TLM model and the QEMU virtual machine is done at the *transaction level block* granularity.

5.2 Applications

In order to assess our compiler results on the Magali platform, representative parts of the LTE-Advanced protocol were extracted to illustrate the challenges in terms of programmability and dynamicity. Overall description of the LTE-Advanced protocol can be found in [33], with implementation examples in [11, 32]. The implemented test case applications corresponds to *ofdma* and *channel decoder* parts from [33]. The test case applications are represented on Fig. 7b and Fig. 7c, and are described to highlight their potential challenge.

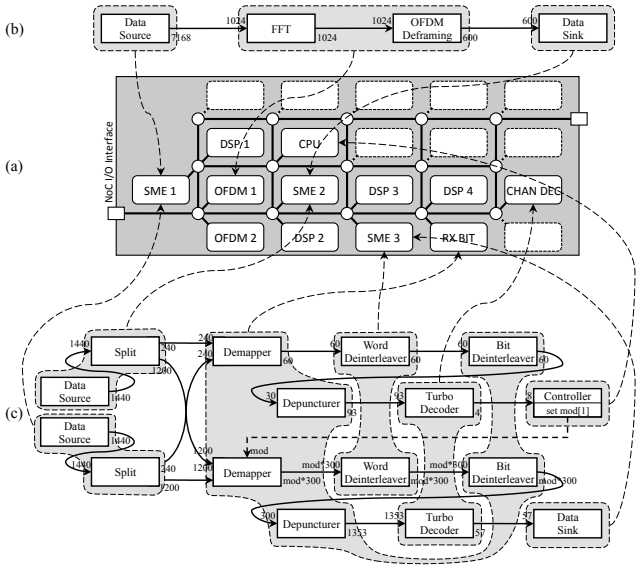


Figure 7. Mapping of the test case applications on Magali.

OFDM *test case* The OFDM test case, presented in Fig. 7b, shows the mapping of the FFT and deframing actors onto a single OFDM core. It abstracts Magali architectural specifics while benefiting from specific hardware operators.

Demodulation test case The demodulation test case is another part of the LTE-Advanced application presented in the lower part of Fig. 7c, in which modulation (the "mod" parameter) would be statically fixed. It illustrates a more complex mapping of actors and communications between 6 blocks.

Parametric Demodulation test case The parametric demodulation (whole Fig. 7c) extends the previous test case by showcasing the use of parameters. Here the "mod" parameter represents the modulation scheme, which depends on the computations done by the upper part of the data-flow — i.e. on the decoding of signaling channels at the beginning of the received frame — and directly impacts the rest of the computation — the decoding of user data in the frame.

5.3 Performances

Application	PaDaF code #lines C++ / time	handwritten code #lines C / ASM / time
FFT	60 / 1 h	150 / 200 / 1 week
demodulation	160 / 4 h	300 / 600 / 1 month
param. demod.	260 / 8 h	500 / 800 / 3 months

Table 1. Comparison of compiled code and handwritten code targeting Magali.

The benefits of using our compiler are described in Tab. 1. Results are based on Magali developers' experience. Required time to write an application is a subjective metric, because its process includes reflection times difficult to gauge, and because it is highly dependent on the developer. However, when applications are written by people of similar technical skills and with the same knowledge of the hardware platform and wireless protocol, it gives a relevant estimate of the benefits coming from the provided tool. Code size for the Magali platform is split between C code for the ARM central controller and assembly code for the distributed control. The rather low *code lines/time* ratio for handwritten code is due to the inherent complexity of programming the platform: distributed control requires configuring different independent hardware blocks with globally consistent values that all together represent the application. Without a dedicated support tool, ensuring — and debugging — this global consistency is an error-prone process for the programmer. As a consequence, whereas the size of the code generated by our compiler is roughly equivalent to the size of handwritten code, the initial code size is divided by five and the development time approximately by 40.

Application	states	transitions	execution time
FFT	1.28×10^4	2.56×10^4	0.1 s
demodulation	2.12×10^6	1.07×10^7	9 s
param. demod.	6.07×10^7	2.22×10^8	199 s

Table 2. Buffer checking results on the SPIN model checker.

Model checking techniques, used for checking buffer requirements, can be limited by complexity issues when exploring large state space. To evaluate this complexity, simulation results using the SPIN model checker are presented in Tab. 2. These simulations were run on a 2.4 GHz Intel Core i5 with 8 Go of RAM running OS X 10.9.2. PROMELA models of the different test cases were generated as described in Section 4.2.1 with all loops unrolled to reduce the number of states tenfold. As expected, the complexity increases with the graph size, but all applications were explored in reasonable time. On Magali, such analysis was not possible before, programmers would profile the code and optimize it if a deadlock was encountered. Using this method, we are now able to prove the absence of deadlock due to communications for this application.

Application	handwritten	[29]	generated	optimized
FFT	149 μs	500 μs (+236%)	168 μs (+13%)	149 μs (+0%)
demod.	180 μs	-	283 μs (+57%)	180 μs (+0%)
parametric demod.	419 μs	-	558 μs (+33%)	288 μs (-31%)

Table 3. Performance result of generated code with respect to handwritten code

The performance for the applications described in section 5.2 are presented in Table 3. The manual — and time consuming — porting of the 3GPP LTE-Advanced application that has been made on Magali has also previously been explained [11]. This approach is used as a baseline to compare our solutions. Previous work on a *radio virtual machine* [29] shows the penalty of not using the distributed control mechanisms, with an overhead of 236% for the simple FFT application. Results of our compilation flow are presented on the *generated* approach, which corresponds to unmodified code generated by our compiler, and the *optimized* approach, with manual optimization to the central controller code generated by the compiler.

The overhead of the *generated* approach vary from 13% for small applications up to 57%, and is due to the central controller latency. This latency is caused by the use of one thread per core which increases the number of synchronizations between the cores and the central controller. The optimized approach uses only one thread, with synchronizations limited to parameter changes. This optimisation was done manually by modifying C code, and should be automated in the future. As a conclusion to these experiments, our compiler produces codes whose performances are similar to the handwritten code for non parametric applications, and even improved for parametric applications.

6. Related Work

Various compilation flows are used to program SDR platforms, many of them programmed using more than one language (C and assembly code, or Matlab and VHDL for instance). On the other hand, many Integrated Design Environments (IDEs) are emerging, targeting general purpose applications on parallel architectures or dedicated to software defined radio. Among these design tools, one can mention OSSIE [18] (implementing SCA), SPEX [26] or DiplodocusDF [19] (see [14] for a complete survey).

Up to now, few SDR programming environment has been adapted to more than one hardware architecture, except for "real" software projects such as GNUradio[2]. GNUradio is adapted to low-performance computing power radio applications but cannot address demanding applications such as LTE-Advanced in real-time. MAPS [10] proposes to program several SDR platforms using a library of so called *nuclei*, which are computational kernels common to several communication standards. These *nuclei* are platform independent and can be implemented in several ways, called *flavors*. Our approach is similar, with a standard API offering computational kernels implemented on different platforms.

Many SDR programming environments are adopting the dataflow MoC. Some MoCs hold much information, offering various levels of static verification and optimization, such as Synchronous data flow (SDF). Others allow very dynamic behaviors, such as Kahn Process Networks (KPN), see [14, 21] for recent surveys. Recently, the need for verifiable but still flexible dataflow MoCs lead to the appearance of two new MoCs: Scenario-Aware DataFlow [30] and

Parametric DataFlow [16]. Fradet et al. identify a subclass of this MoC called *Schedulable Parametric DataFlow* (SPDF) where the schedulability of the dataflow graph can still be assessed statically, which matches the kind of dynamicity that is required in telecom applications such as LTE-Advanced [7]. They also provide a scheduling procedure for SPDF graphs and are able to check FIFO size, but they do not reach the level of accuracy obtained by micro-schedules because parameter setting are always included after actor's execution.

In terms of language, the only other dataflow language providing complex graph construction we are aware of is ΣC [20]. It is based on an extension of C, with a complete new compilation flow. In contrary, our approach is based on existing compilation tools, both requiring less engineering effort and providing all the C++ expressivity for the DFG construction.

Scheduling for buffer minimization is a NP complete problem [8]. Many heuristics have been developed to schedule under memory constraints [8, 13, 17, 23]. We focus on model-checking solutions based on the work by Geilen et al. [17], which solves the scheduling problem on constrained buffer size for synchronous dataflow graphs. Using a similar approach, Damavandpeyma et al. [13] minimize buffer on scheduled synchronous dataflow graphs. In this context, the originality of our work is twofold, with the modelization of *parametric* dataflow graphs, as well as the use of a finer grain modelization using micro-schedule to check the absence of deadlock on scheduled dataflow graphs.

7. Conclusion

As dynamicity increases in applications, new models of computation such as parametric dataflow are used in the signal processing domain. In this paper, we propose a new compilation flow, based on the LLVM framework, that compiles SPDF graphs down to heterogeneous MPSoC. We also propose a new format based on C++ to express complex parametric graphs. In order to solve practical problem encountered during code generation for heterogeneous MPSoC, we introduce the micro-schedule formalism to describe actors communication behavior. Based on this formalism, precise buffer size verification can be performed.

To validate our results, experimentation on the Magali platform are performed using the LTE-Advanced application. All test cases are successfully checked for their buffer usage. Performances close to the handwritten implementation are generated automatically by our compiler (max +57%). Support for other heterogeneous MPSoC is another important future work to prove the portability of our approach.

References

[1] clang: a C language family frontend for LLVM. URL `http://clang.llvm.org`. 2014.

[2] GNU radio framework. URL `http://gnuradio.org`. 2014.

[3] The LLVM Compiler Infrastructure. URL `http://llvm.org`. 2014.

[4] Spin - Formal Verification. URL `http://spinroot.com`. 2014.

[5] V. Bebelis, P. Fradet, A. Girault, and B. Lavigueur. A Framework to Schedule Parametric Dataflow Applications on Many-Core Platforms. In *17th workshop on Compilers for Parallel Computing, CPC*, Lyon, FR, July 2013.

[6] V. Bebelis, P. Fradet, A. Girault, and B. Lavigueur. BPDF: A statically analyzable dataflow model with integer and boolean parameters. In *Proc. International Conference on Embedded Software (EMSOFT)*, pages 1–10, Montreal, QC, Sept. 2013.

[7] H. Berg, C. Brunelli, and U. Lucking. Analyzing models of computation for software defined radio applications. In *Proc. International Symposium on System-on-Chip*, pages 1–4, Tampere, Finland, Nov. 2008.

[8] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee. Synthesis of embedded software from synchronous dataflow specifications. *Journal of VLSI signal processing systems for signal, image and video technology*, 166(2):151–166, 1999.

[9] João M. P. Cardoso, P. C. Diniz, and M. Weinhardt. Compiling for reconfigurable computing. *ACM Computing Surveys*, 42(4):1–65, June 2010.

[10] J. Castrillon, S. Schürmans, A. Stulova, W. Sheng, T. Kempf, R. Leupers, G. Ascheid, and H. Meyr. Component-based waveform development: The Nucleus tool flow for efficient and portable software defined radio. *Analog Integrated Circuits and Signal Processing*, 69(2-3):173–190, June 2011.

[11] F. Clermidy, R. Lemaire, X. Popon, D. Ktenas, and Y. Thonnart. An Open and Reconfigurable Platform for 4G Telecommunication: Concepts and Application. In *12th Euromicro Conference on Digital System Design, Architectures, Methods and Tools*, pages 449–456, Patras, Greece, Aug. 2009.

[12] F. Clermidy, C. Bernard, R. Lemaire, J. Martin, I. Miro-Panades, Y. Thonnart, P. Vivet, and N. Wehn. A 477mW NoC-based digital baseband for MIMO 4G SDR. In *Proc. IEEE International Solid-State Circuits Conference, ISSCC*, pages 278–279, San Francisco, CA, Feb. 2010.

[13] M. Damavandpeyma, S. Stuijk, T. Basten, M. Geilen, and H. Corporaal. Modeling static-order schedules in synchronous dataflow graphs. In *2012 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 775–780, Dresden, Germany, Mar. 2012.

[14] M. Dardaillon, K. Marquet, J. Martin, T. Risset, and H.-P. Charles. Cognitive Radio Programming: Existing Solutions and Open Issues. Technical Report September, Inria, 2013.

[15] M. Dardaillon, K. Marquet, T. Risset, J. Martin, and H.-p. Charles. Compilation for heterogeneous SoCs : Bridging the gap between software and target-specific mechanisms. In *Workshop on High Performance Energy Efficient Embedded Systems (HIPEAC)*, Vienna, Austria, Jan. 2014.

[16] P. Fradet, A. Girault, and P. Poplavko. SPDF: A schedulable parametric data-flow MoC. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 769–774, Dresden, Germany, Mar. 2012.

[17] M. Geilen, T. Basten, and S. Stuijk. Minimising buffer requirements of synchronous dataflow graphs with model checking. In *Proc. 42nd annual conference on Design automation - DAC '05*, page 819, San Diego, CA, June 2005.

[18] C. Gonzalez, C. Dietrich, S. Sayed, H. Volos, J. Gaeddert, P. Robert, J. Reed, and F. Kragh. Open-source SCA-based core framework and rapid development tools enable software-defined radio education and research. *IEEE Communications Magazine*, 47(10):48–55, Oct. 2009.

[19] J. Gonzalez-Pina, R. Ameur-Boulifa, and R. Pacalet. DiplodocusDF, a Domain-Specific Modelling Language for Software Defined Radio Applications. In *38th Euromicro Conference on Software Engineering and Advanced Applications*, pages 1–8, Cesme, Izmir, Sept. 2012.

[20] T. Goubier, R. Sirdey, S. Louise, and V. David. ΣC A Programming Model and Language for Embedded Manycores. In *Algorithms and Architectures for Parallel Processing - 11th International Conference, ICA3PP*, pages 385–394, Melbourne, Australia, Oct. 2011.

[21] W. M. Johnston, J. R. P. Hanna, and R. J. Millar. Advances in dataflow programming languages. *ACM Computing Surveys*, 36(1):1–34, Mar. 2004.

[22] S.-H. Kang, H. Yang, L. Schor, I. Bacivarov, S. Ha, and L. Thiele. Multi-objective mapping optimization via problem decomposition for many-core systems. In *Embedded Systems for Real-time Multimedia (ESTIMedia), 2012 IEEE 10th Symposium on*, pages 28–37, Oct 2012.

[23] M. Karczmarek, W. Thies, and S. Amarasinghe. Phased scheduling of stream programs. In *Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'03)*, page 103, San Diego, CA, June 2003.

[24] S. Kwon, Y. Kim, W. C. Jeun, S. Ha, and Y. Paek. A retargetable parallel-programming framework for mpsoc. *ACM Trans. Des. Autom. Electron. Syst.*, 13(3):1–18, 2008.

[25] E. A. Lee and D. G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, June 1987.

[26] Y. Lin, R. Mullenix, M. Woh, S. Mahlke, T. Mudge, A. Reid, and K. Flautner. SPEX: A programming language for software defined radio. In *SDR Forum Technical Conference*, pages 13 – 17, Orlando, Florida, Nov. 2006.

[27] K. Marquet and M. Moy. PinaVM: a SystemC Front-End Based on an Executable Intermediate Representation. In *Proceedings of the tenth ACM international conference on Embedded software - EMSOFT '10*, page 79, Scottsdale, Arizona, Oct. 2010.

[28] P. R. Panda. SystemC - A modeling platform supporting multiple design. In *Proc. 14th International Symposium on Systems Synthesis (ISSS)*, pages 75–80, Montreal, QC, Sept. 2001.

[29] T. Risset, R. Ben Abdallah, A. Fraboulet, and J. Martin. *Digital Front-End in Wireless Communications and Broadcasting*, chapter Programming models and implementation platforms for software defined radio configuration, pages 650–670. Cambridge University Press, 2011. ISBN 9781107002135.

[30] S. Stuijk, M. Geilen, B. Theelen, and T. Basten. Scenario-aware dataflow: Modeling, analysis and implementation of dynamic applications. In *International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation*, pages 404–411, Samos, Greece, July 2011.

[31] M. Woh, Y. Harel, S. Mahlke, T. Mudge, C. Chakrabarti, and K. Flautner. SODA: A Low-power Architecture For Software Radio. In *33rd International Symposium on Computer Architecture, ISCA*, pages 89–101, Boston, MA, June 2006.

[32] M. Woh, S. Seo, H. Lee, Y. Lin, S. Mahlke, T. Mudge, C. Chakrabarti, and K. Flautner. The next generation challenge for software defined radio. In *Embedded Computer Systems: Architectures, Modeling, and Simulation*, pages 343–354, Samos, Greece, 2007.

[33] J. Zyren and W. McCoy. Overview of the 3GPP long term evolution physical layer. Technical report, Freescale Semiconductor Inc., 2007.