A New Compilation Flow for Software-Defined Radio Applications on Heterogeneous MPSoCs

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

A New Compilation Flow for Software Defined Radio Applications on Heterogeneous MPSoC

MICKAEL DARDAILLON, Université de Lyon, INRIA, INSA-Lyon, CITI-INRIA
KEVIN MARQUET, Université de Lyon, INRIA, INSA-Lyon, CITI-INRIA
TANGUY RISSET, Université de Lyon, INRIA, INSA-Lyon, CITI-INRIA
JEROME MARTIN, Univ. Grenoble Alpes, CEA, LETI, Minatec campus
HENRI-PIERRE CHARLES, Univ. Grenoble Alpes, CEA, LIST, Minatec campus

The advent of portable Software Defined Radio (SDR) technology is tightly linked to the resolution of a difficult problem: efficient compilation of signal processing applications on embedded computing devices. Modern wireless communication protocols use packet processing rather than infinite stream processing and also introduce dependencies between data value and computation behaviour leading to dynamic dataflow behavior. Recently, parametric dataflow has been proposed to support dynamicity while maintaining the high level of analyzability needed for efficient real life implementations of signal processing computations.

This paper presents a new compilation flow that is able to compile parametric dataflow graphs. Built on the LLVM compiler infrastructure, the compiler offers an actor-based C++ programming model to describe parametric graphs, a compilation front end for graph analysis, and a back end which currently matches the Magali platform: a prototype heterogeneous MPSoC dedicated to LTE-Advanced. We also introduce an innovative scheduling technique, called micro-scheduling, allowing to adapt the mapping of parametric dataflow programs to the specificities of the different possible MPSoCs targeted. A specific focus on FIFO sizing on the target architecture is presented. The experimental results show compilation of 3GPP LTE-Advanced demodulation on Magali with tight memory size constraints. The compiled programs achieve performances similar to hand written code.

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

General Terms: Algorithms, Languages

Additional Key Words and Phrases: dataflow, programming model, heterogeneous MPSoC, compiler, scheduling, Software Defined Radio

ACM Reference Format:
Mickaël Dardaillon, Kevin Marquet, Tanguy Risset, Jérôme Martin and Henri-Pierre Charles, 2015. A New Compilation Flow for Software Defined Radio Applications on Heterogeneous MPSoC *ACM Trans. Architec. Code Optim.* 0, 0, Article 0 (February 2016), 25 pages.
DOI : http://dx.doi.org/10.1145/0000000.0000000

Part of this work was published as M. Dardaillon, K. Marquet, T. Risset, J. Martin and H.-P. Charles. A Compilation Flow for Parametric Dataflow: Programming Model, Scheduling, and Application to Heterogeneous MPSoC. *International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, New Delhi, India, Oct. 2014. This work extends previous publication by: (i) Providing a thorough presentation of a new front end technique, (ii) Introducing a new Promela model for buffer size verification and (iii) Updated results reflecting the subtantial improvements provided by the new Promela model.
This work is sponsored by Région Rhône Alpes ADR 11 01302401. Author's addresses: M. Dardaillon, K. Marquet and T. Risset (corresponding author), INSA-Lyon, CITI-Inria, F-69621 Villeurbanne, France; email: tanguy.risset@insa-lyon.fr; J. Martin, CEA, LETI, Minatec campus, F-38054 Grenoble, France; H.-P. Charles, CEA, LIST, Minatec campus, F-38054 Grenoble, France.

DOI : http://dx.doi.org/10.1145/0000000.0000000

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, as they allow both static analysis and explicit parallelism, and are 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, but also energy efficient to be embedded in mobile phones.

The advent of "Advanced" 4G (e.g. LTE-Advanced) and forthcoming 5G wireless protocols, as well as the development of software defined radio (SDR) technologies and cognitive radio networks reveal new challenges for expressing and compiling wireless applications. The physical layer of these wireless protocols has a dynamic behavior and requires fast dynamic reconfigurations which are not possible with today's wireless devices. These technological trends have re-activated past research areas such as dynamic dataflow compilation or hardware implementation of signal processing algorithms. In particular, the need for flexible but still verifiable programs has led 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 (i.e. QPSK, 16-QAM, etc.) used to decode samples in a LTE-Advanced frame 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 [Berg et al. 2008; Wiggers 2009] because some data-dependent behavior appear. However such examples rarely occur and usually do not necessarily require a complete dynamic dataflow model of computation.

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 [Woh et al. 2006; Clermidy et al. 2010]) 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. Some implementations of LTE-Advanced being commercially deployed already exists, but 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:

— A new compilation flow for the SDR platforms. Our compilation framework was instantiated for the Magali [Clermidy et al. 2009b] architecture. Magali programs are usually tuned by hand, our compilation flow generates compiled programs whose performances are equivalent to manually tuned programs performances. Our compiler provides an innovative front end that builds, analyses and generates an internal representation for parameterized dataflow graph described in C++ as well as a back end dedicated to the Magali MPSoC.

— A new high level format for expressing parametric dataflow graphs in reduced form. This format permits to express dataflow graphs in a parametrized high level programming model. For example, it can describe a MIMO receiver with N antennas and construct the extended graph by setting the value of N at compilation time.

— An efficient static analysis paradigm called *micro-schedule* that permits a more precise analysis of deadlock when mapping parametrized dataflow graph to real ar-

chitecture. We also present an improved model checking use to the specific problem of advanced actor pipelining in the context of dedicated target architecture with small buffers.

The paper is organized as follows: Section 2 presents the context of the work, Magali target architecture, and the specificities of modern wireless waveform which motivates the development of a new programming paradigm. Section 3 presents our compilation framework. Section 4 present parametric dataflow scheduling and shows why the existing scheduling techniques are not adapted to the Magali target. We introduce the micro-scheduling refinement in Section 5 and show an efficient way to check that buffer sizes available on the target architecture are adapted to a given schedule. Evaluation of the compilation flow is presented in Section 6 and related works are presented in Section 7.

2. EXPERIMENTAL CONTEXT: PLATFORMS AND APPLICATIONS

Many recent and forthcoming communication protocols will need flexibility in radio resource handling. We mentionned above fast dynamic reconfiguration needed in LTE-Advanced and this is obviously also true for cognitive radio applications. Moreover, this flexibility also concerns 5G protocols which will be faced with spectrum saturation, as well as internet of things and machine to machine communications which must adapt to the rapid evolution of standards. These different levels of flexibility are enabled by the Software Defined Radio technology. Much R&D effort has been dedicated by the main radio communication industrial actors to provide efficient architectures for SDR, however programming such complex machines is still a research challenge and a bottleneck for product development.

The reason is that SDR technology uses a wide variety of execution models: homogeneous and heterogeneous multi-cores such as commercial baseband processors from companies like Texas Instrument, Qualcomm or Freescale; FPGA-based machines; dedicated ASIC; cloud-RAN architectures, etc. Moreover, there is no consensus on the best programming model for programming flexible radio protocols: traditional dataflow models are widely used but must be adapted to very fast reconfiguration needed by recent protocols (LTE-Advanced for instance) and cognitive radio capabilities.

Today, SDR programmers are missing several tools, either for expressing high-level SDR programs (waveforms with real-time constraints), or for mapping them onto existing parallel SDR architectures. Beyond this, SDR requires us to rethink the full software stack: operating system, virtualization mechanisms, middleware for over-the-air programming, etc. Therefore, for industrial and large-scale applicability of new wireless technologies, it is urgent to invest in the software infrastructure for radio programming. This work proposes a step in that direction with a domain-specific compiler that can take into account the characteristics of the platforms targeted and of the waveform applications.

2.1. Magali Hardware Architecture

The Magali chip [Clermidy et al. 2009b], represented on Fig. 1 is a system on chip (SoC) dedicated to physical layer processing of OFDMA radio protocols, with a special focus on 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 the programming of sequences of computations for each block, thus limiting the required number of reconfigurations

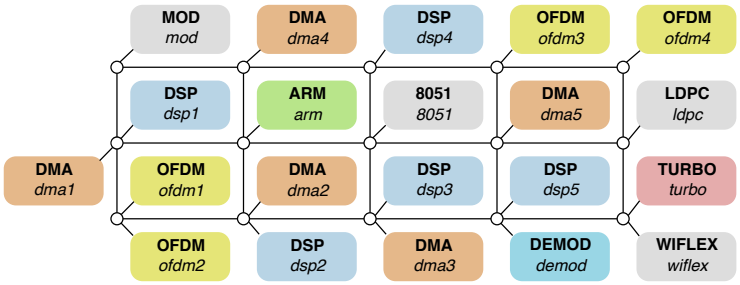


Fig. 1: Magali hardware architecture used for our experimentations.

done by the CPU in the case of complex applications. These distributed control sequences, called *contexts*, were very difficult to write by hand in a coherent way for all IPs (general purpose processors, DSP or dedicated ASICs) and hence are one of the main motivation to write a compiler for Magali.

As it is often the case for the complex MPSoC, program validation for Magali is done on a dedicated *simulation framework* that provides functional simulation of the IPs with accurate performance estimation. The Magali simulation framework is based on a SystemC *transaction level model* (TLM) of the Magali chip. Timing details are extracted from the blocks synthesized in 65nm CMOS technology. The ARM central controller code runs 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. The chip was manufactured in 2010 and used as a demonstration for LTE-Advanced applications [Clermidy et al. 2010].

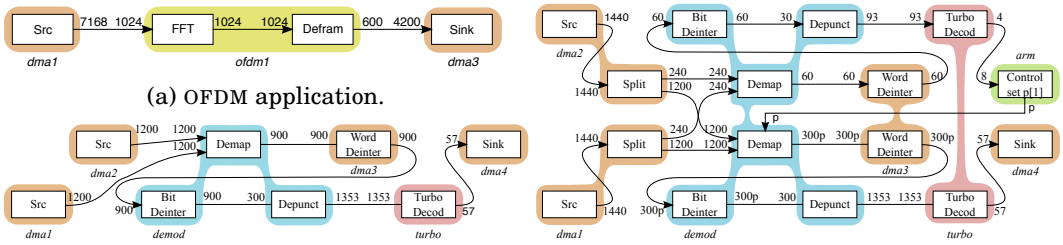
2.2. New Wireless Waveform Application Constraints

The LTE-Advanced protocol used to validate our compilation flow presents several characteristics of modern wireless waveforms which indeed correspond to difficult problems to solve for designers. The first problem comes from the complexity of the protocols: they include many blocks in parallel as illustrated in Fig 2 which shows parts of the PHY layer of a LTE reception signal with two antennas. This problem is at the heart of SDR compilation and includes in particular: the mapping problem, scheduling problem and communication handling. Many designers (but not all of them) have chosen to use dataflow format to express the waveforms to ease parallelism expression.

The second and most difficult problem concerns the dynamicity of these new waveforms. A typical adaptative transmission will adapt its decoding (and hence its rate) to the transmission conditions. This might imply changing the modulation on one channel but can go up to a complete reconfiguration within a frame. All these reconfigurations have to be done within approximately 1 ms. This fast dynamic reconfiguration motivates the building of dedicated chip such as Magali and the use of parameterized data flow computation model as we propose in section 4. In our proposal, parameters are used to configure decoding blocks and are dependent on values of the dataflow itself.

2.3. LTE-Advanced 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 [Zyren and McCoy 2007], with implementation examples in [Woh et al. 2007; Clermidy et al. 2009b]. The implemented test case applications correspond to *ofdma*



(a) OFDM application.

(b) Demodulation application.

(c) Parametric demodulation application.

Fig. 2: Applications compiled on Magali, the colors indicate the mapping of actors on IPs.

and *channel decoder* of LTE-Advanced. The test case applications are represented on Fig. 2a, Fig. 2b and Fig. 2c, and are described hereafter.

OFDM *test case.* The OFDM test case, presented in Fig. 2a, shows the mapping of the FFT and deframing actors onto a single OFDM core. It is used to prove that our compiler can take advantage of the very specific hardware mechanisms of Magali's IPs, the so called *contexts* mentionned in section 2.1.

Demodulation test case. The demodulation test case is another part of the LTE-Advanced application presented on Fig. 2b. It illustrates a more complex mapping of actors and communications between 6 blocks and proves that inter-IP communications are also handled efficiently by our compiler.

Parametric Demodulation test case. The parametric demodulation (Fig. 2c) extends the previous test case by showcasing the use of parameters. Here the parameter "p" represents the modulation scheme, which depends on the computations done by the upper part of the dataflow — 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. This application presents, to the best of our knowledge, the first compilation of a parametric dataflow program on a real heterogeneous MPSoC platform.

3. COMPILATION FRAMEWORK

In this Section, we describe simultaneously the input format used to express parametric dataflow applications and the compilation flow that we have built to compile these applications. A dataflow compilation framework should compile high level specifications of a DataFlow Graph (DFG) representation (we use the parametric dataflow paradigm) and produce executable code needed to program the target platform. To be retargetable, it should also take as input a description of the target architecture.

Our dataflow compilation framework, illustrated in Fig. 3, is split into two phases: i) *Front End* for parsing and analysis of the DFG, is introduced in Section 3.2, and ii) *Back End* for mapping, scheduling and code generation is described in Section 3.3.

Our compilation also includes features that are not present simultaneously in other dataflow compilation frameworks: handling of parametric dataflow applications, complex DFG construction, buffer size checking on scheduled application, and code generation for complex heterogeneous SoCs. We start by introducing the format used as input for our compilation flow in the next Section.

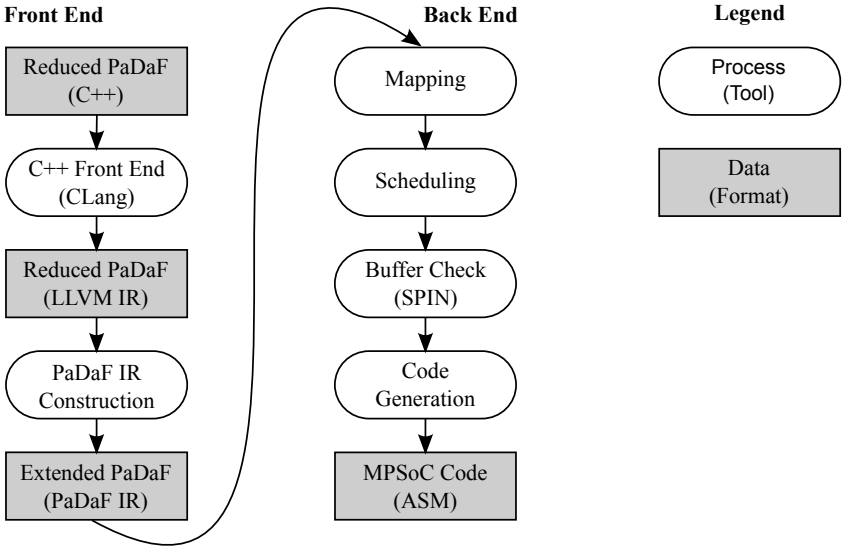


Fig. 3: Proposed compilation flow from (parametric) dataflow to heterogeneous SoC.

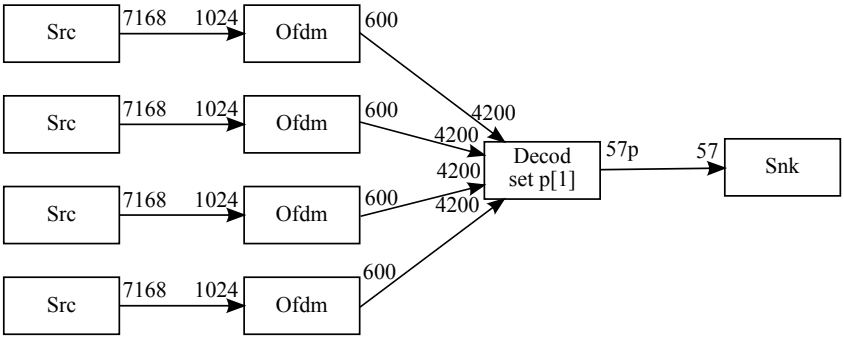


Fig. 4: SPDF application graph for a MIMO receiver with 4 antennas.

3.1. Parametric Dataflow Format: PaDaF

The input format follows the *Schedulable Parametric DataFlow* (SPDF) model of computation proposed by Fradet et al. [2012]. The SPDF graph of Fig. 4 illustrates a MIMO receiver with 4 antennas. The `set p[1]` in actor `Decod` indicates that actor `Decod` produces a new integer value for parameter `p` each time it fires. Also, the graph indicates that actor `Decod` defines `p` and then output `57p` tokens (the reader should refer to [Fradet et al. 2012] for a detailled presentation of SPDF).

The input format we propose, called PaDaF (*Parametric Dataflow Format*), allows us to describe both actors behaviour, in C++, and a parametric DFG using the actors. Fig. 5a and 5b present actors declaration of the program implementing the MIMO receiver SPDF graph. They illustrate how actors are declared and show specific classes for data ports (`PortIn` and `PortOut` classes) and for parameter ports (`ParamOut` class). The constructor of each actor simply specifies the number of tokens for each port of the actor as illustrated on the constructor of `Ofdm` class. However, specifying that number can be more complex because (i) this number might be parametric (see output port of

```
class Ofdm : public Actor {
    PortIn<int> Iin;
    PortOut<int> Iout;
    void compute();
    Ofdm():
      Iin(1024), Iout(600){}
};
```

(a) Actor OFDM declaration.

```
class Decod : public Actor {
  std::vector<PortIn<int>*> Iin;
  PortOut<int> Iout;
  ParamOut p;
  void compute();
  int nbAnt;
  Decod(int nb) : p(1), Iout(p*57) {
    nbAnt = nb;
    for(int i=0; i<nbAnt; i++)
      Iin[i] = new PortIn<int>(4200);
  }
```

(b) Actor Decod declaration.

Fig. 5: Actors OFDM and Decod declaration in PaDaF.

```
Src src[NB_ANT];
Ofdm fft[NB_ANT];
Decod 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) MIMO receiver DFG in PaDaF.

```
void Decod::compute() {
  [...]
  int coef = nbAnt * var;
  for(int i=0; i<nbAnt; i++) {
    val[i] = Iin[i]->pop();
    [...]
  }
  p.set(size);
  Iout.push(res, 57*size);
}
```

(b) Actor Decod `compute()` method

Fig. 6: `compute()` method and reduced graph expression of the graph of Fig. 4 in PaDaF.

`Decod` actor), and (ii) as the number of port of an actor might be symbolic (e.g. `nbAnt` input ports for `Decod` actor), we might need some code to specify the number of tokens on each port as shown on Fig. 5b. This format is close to systemC [Panda 2001], using C++ to support a specific model of computation.

The originality of PaDaF is that it permits to describe the DFG in a *closed form* (or *reduced graph*), i.e. as a sequence of instructions describing how to build the graph. This sequence can use C++ control structure instructions (**for** loop for instance) and any C++ structure for that matter. All information needed to *construct* the graph have to be known at compilation time. Fig. 6a illustrates the use of PaDaF to describe, in a closed form, an SPDF graph with a symbolic number of `Ofdm` nodes (`NB_ANT` is the number of antennas). The *extended* DFG of this PaDaF application is the MIMO receiver of Fig. 4 for `NB_ANT`=4.

Each actor has a single `compute()` method which is executed at each firing 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 `Decod` actor is shown on Fig. 6b.

Choosing C/C++ language for the core code of the actors offers 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;

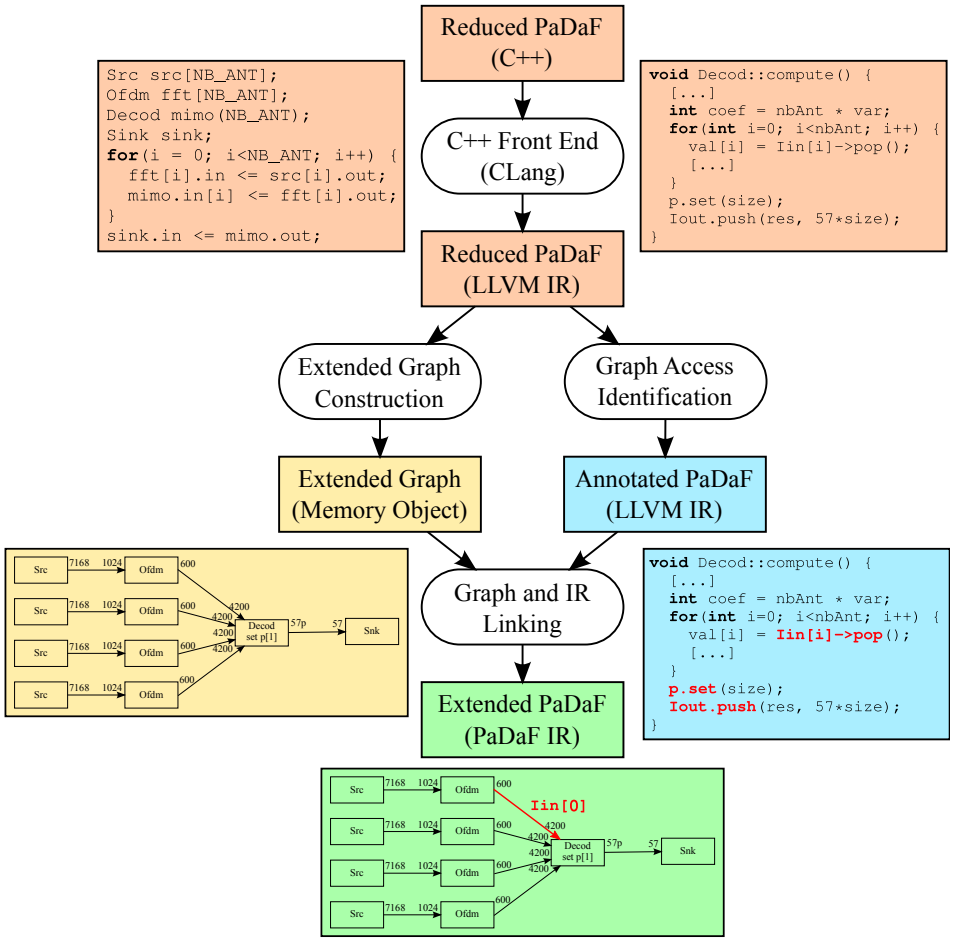


Fig. 7: Details of the front end compilation flow with C++ examples for clarity.

it provides important capabilities such as the ability to iterate for the construction of complex structures (channels of the MIMO receiver illustrated on Fig. 6a).

3.2. Compiler Front End

This Section deals with the front end of our compiler that generates the *Intermediate Representation* (IR) and builds the extended DFG of the application described. The DFG construction consitutes in itself an original contribution of this work as it implies IR analysis at compilation time, as well as unconventional IR *execution at compilation time*. It is derived from PinaVM [Marquet and Moy 2010], a front end for SystemC analysis. To understand the potential complexity of the DFG construction, we use the MIMO graph construction code of Fig. 6a. This construction implies a **`for`** loop to link actors `src` to `fft`, and `fft` to `mimo`.

Fig. 7 introduces a detailed view of this DFG construction. The compilation flow is based on the LLVM compiler infrastructure [llv 2015]. Our input format PaDaF being based on standard C++, it takes advantage of LLVM C++ front end, Clang [cla 2015], to generate the LLVM IR. The front end illustrated in Fig. 7 is decomposed in 3 steps:

(i) construction of the extended graph, presented in Section 3.2.1, (ii) identification of graph access methods in the `compute()` methods, introduced in Section 3.2.2 and (iii) linking of the graph access methods to their corresponding edge in the extended graph is described in Section 3.2.3.

3.2.1. Extended Graph Construction. The first compilation step is to construct the DFG *in memory*, this step resembles the elaboration step in architecture description languages such as SystemC or VHDL. It aims at building the extended graph.

Our technique uses an *execution* of the graph construction code *at compile time*. This execution instanciates all actors and connects them together. The result is a set of C++ objects instanciated in memory. The compiler uses this graph representation in memory for the remaining of the compilation flow.

This step is carried out by the just-in-time (JIT) compiler of the LLVM compiler framework. The JIT compiler executes the graph construction code with an added callback function at the end of the code to access the graph constructed in memory within the compiler. This memory representation includes each instanciated actor and a link to their `compute()` method in LLVM IR. It also includes the edges between actors, as well as their production and consumption rates either static or parametric, in symbolic form. Once the graph is built, the problem is to link this extended graph with the `push/pop` operations in the `compute()` methods. this is described hereafter.

3.2.2. Graph Access Identification. In each `compute()` method, data consumption and production are done on actor's ports which are connected to edges of the extended graph. More precisely, they are performed by methods (*e.g.* `push()`) of each data ports object, as presented in Section 3.1. The first step is to isolate these method calls in the LLVM IR.

In order to find data access calls, the `compute()` method is scanned and each function call is assessed. This evaluation is based on method names (i.e. `push`, `pop` for data and `set`, `get` for parameter) which appear as mangled in the LLVM IR. Each call is annotated with a metadata indicating its role (e.g. access data, produce parameter) in the LLVM IR.

3.2.3. Graph and IR Linking. Once all the methods calls are found, the last step is to link these accesses to their corresponding edges in the extended graph, reminding that this extended graph has been built in memory in the first step (see Section 3.2.1). We chose to identify each accessed port with its address in memory. In general this address is difficult to compute as the code used to access the ports might be arbitrarily complex. For instance, on Fig. 6b, access to each input port by `Decod` actor is done through the `Iin[i]` expression, `i` might itself be the result of another expression. Rather than analysing this code statically, we propose to compute the address of port `Iin[i]` just as we constructed the graph, by *executing only* the code computing this address.

The `compute()` method contains many instructions, only few of them are used to compute the address of the accessed port. Using *slicing* [Marquet and Moy 2010], we execute only the instructions required for the computation of this address. For instance, the computation of the port address `Iin[i]` in Fig. 6b is not dependent on the value of the `coef` variable, but only on the value of variable `i`.

LLVM IR has a Static Single Assignment form (SSA), which eases the analysis of the control flow defining the dependencies between instructions. The algorithm marks all the instructions useful to compute the variables of interest, and these instructions are placed in a new function. This function takes as argument the actor to which the analyzed `compute` method belongs. The execution of this function by the LLVM JIT returns the address of the port in the graph. In this way, each method call is linked to

the accessed port. This information is contained in the metadata of the method call, and used in the remaining compilation flow.

The main limitation of this method for the graph construction is the assumption of a static graph architecture. This constraint matches the dataflow model of computation, in which the structure of the DFG is known at compilation and no actor or edge creation is allowed at execution time.

3.3. Compiler Back End

Once we have built the extended graph of a dataflow application, the back end of the compiler is in charge of specializing the application 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 [Cardoso et al. 2010; Castrillon et al. 2011; Kwon et al. 2008; Kang et al. 2012], and is still a very active research area because satisfactory solutions are hard to develop. 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. We also assume that the mapping is static, i.e. there is no task migration.

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 [Clermidy et al. 2009b] does not support runtime scheduling. In such cases, we generate a static schedule for the execution of the different actors on the core. The scheduling methodology will be described in Section 4, introducing in particular the micro-scheduling technique.

Buffer Checking. Given the very harsh platform constraints (e.g. static scheduling, memory constraints), we introduce a buffer size verification step, using a model checking technique, before code generation. This verification 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 and will be explained in Section 5. This model controls the absence of deadlock due to memory constraints, as requested by Magali programmers that could not foresee deadlock situation due to memory size before this work. Evaluation of the verification step on several applications extracted from LTE is presented Section 5.2.

Code Generation. The code generation proposed 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 2, 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 schedules are managed locally by the cores. This approach differs from classic telecommunication control, where applications are split into different *phases*, each one running a static dataflow, whereas *phase* transitions reflect parameter changes [Risset et al. 2011]. By relaxing the control constraints, we aim to take advantage of the potential pipelining introduced by the dataflow model of computation. Evaluation of our compiler in terms

of development time and generated code performance for the Magali platform is done in Section 6.

4. PARAMETRIC DATAFLOW SCHEDULING

Scheduling is a key optimisation problem for the efficient mapping of dataflow applications on a real hardware. In this Section, we show how the well-known case of static dataflow scheduling has been recently extended to parametric dataflow, bringing more flexibility in the use of the dataflow model. We also show current limitations of these scheduling techniques when targeting real hardware platforms.

4.1. Scheduling Static Dataflows

Dataflow languages rely on a model of computation (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 $e = (A_1, A_2) \in \mathcal{E}$ represents a FIFO buffer from its source actor A_1 to its destination actor A_2. The execution (or firing) of an actor A 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. DFGs 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 A is an atomic operation.

Many dataflow-compliant programming models have been proposed for specific applications [Wiggers 2009]. An important category comprises dataflows where the graph topology and rates are static, i.e. fixed and known at compile-time. A famous example of such static dataflow representation is called *Synchronous DataFlow* (SDF [Lee and Messerschmitt 1987]). 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 verifies these properties with the minimum number of firing of each actor is called an *iteration*, it can be obtained by solving the so-called system of balance equations. This system is made of one equation per edge $e = (A_1, A_2)$ of the form:

$$\#A_1 \cdot r_{e,1} = \#A_2 \cdot r_{e,2} \tag{1}$$

where $\#A_1$ and $\#A_2$ denote the number of firings of the actors A_1 and A_2 in an iteration, $r_{e,1}$ is the output rate of A_1 on edge e, and $r_{e,2}$ is the input rate of A_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) [Lee and Messerschmitt 1987].

4.2. 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 7. Among them, SPDF has shown interesting properties, being used to program homogeneous multi-core architectures [Bebelis et al. 2013a] as well as heterogeneous SoCs [Dardaillon et al. 2014b].

SPDF [Fradet et al. 2012] 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,

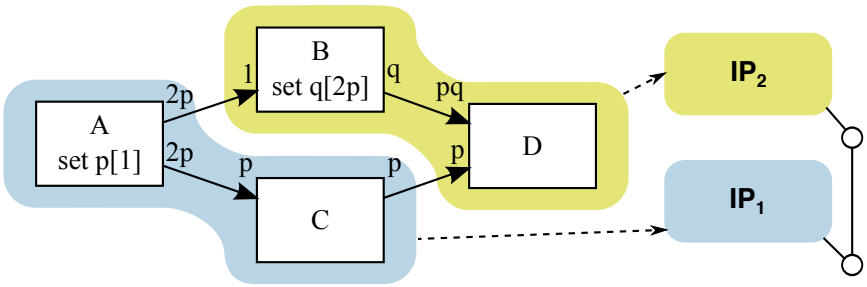


Fig. 8: Example of dataflow application in the SPDF model of computation with its mapping on a two-IP MPSoC.

or products of these two. The reader should refer to [Fradet et al. 2012] for a more formal definition of SPDF.

Fig. 8 reports on the left 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 not imply a sequential ordering of the firings. A scheduling algorithm that computes this vector is presented in [Fradet et al. 2012].

A parameter cannot change anywhere 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 [Bebelis et al. 2013a; Bebelis et al. 2013b], to impose that the parameters change only once per iteration.

Using the $AB^{2p}C^2D^2$ notation for the iteration vector does not indicate when and where parameters are set and used. Fradet et al. [2012] use the term *quasi-static schedule* to refer to a schedule in which there are indication about production and comsumption of parameter (in addition to production and consumption of data). Although there has been other semantics associated to the term“quasi-static schedule”, we use the one of Fradet et al.: a quasi-static schedule is a set of elements executed in a sequential manner, these elements are of three kinds:

— Executing n times the actor A. It is denoted A^n, where n can be a parametric expression.
— Actor A getting the value of a parameter p is denoted $get_A(p)$ (or $get(p)$ when it is not ambiguous),
— Actor A setting the value of a parameter p, denoted $set_A(p)$ (or simply $set(p)$).

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. 8, the quasi-static schedule of the graph corresponding to the extension of the iteration vector with parameter synchronization, is schedule (2):

$$\big(A; set_A(p)\big) \quad \big(get_B(p); B^{2p}; set_B(q)\big) \quad \big(get_C(p); C^2\big) \quad \big(get_D(p); get_D(q); D^2\big) \qquad (2)$$

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 and can be extended to cyclic graph on certain conditions [Bhattacharyya et al. 1999]. Finding a sequential quasi-static schedule for a SPDF graph has been studied in [Fradet et al. 2012], hence in this paper we assume that a valid sequential schedule exists for our applications. For instance, schedule (3) is a valid sequential schedule obtained by topological sort of the SPDF graph of Fig. 8:

$$\left(A; set_A(p); get_C(p); C^2; get_B(p); B^{2p}; set_B(q); get_D(p); get_D(q); D^2\right) \tag{3}$$

Such a global sequential schedule of our SPDF graph can be easily used as a starting point for finding a distributed scheduling onto a multi-core platform given a specific mapping such as the one represented on the right of Fig. 8 with two IP cores. In the general case, one or several graph actors may be mapped on a given IP. A distributed 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. 8 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 on schedule (4) a valid multi-core schedule by scheduling on each core, the actors in the order it was scheduled in the sequential schedule:

$$\begin{array}{l} S_{\text{IP}_1} = \left(A; set_A(p); get_C(p); C^2\right) \\ S_{\text{IP}_2} = \left(get_B(p); B^{2p}; set_B(q)\ ; get_D(p); get_D(q); D^2\right) \end{array} \tag{4}$$

If parameters are shared by actors mapped on the same IP, we can remove redundant synchronization. We then obtain schedule (5):

$$\begin{array}{l} S_{\text{IP}_1} = \left(A; set(p); C^2\right) \\ S_{\text{IP}_2} = \left(get(p); B^{2p}; set(q); D^2\right) \end{array} \tag{5}$$

4.3. Limitation of Traditional Dataflow Formalisms for Code Generation

In many works dealing with classical SDF schedule [Lee and Messerschmitt 1987; Geilen et al. 2005; Bhattacharyya et al. 1999], 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 because of the cost and power consumption of memory on a SoC. In embedded hardware platforms for example, memory reserved for data communications between actors is usually very restricted. The Magali platform only allows *16 bytes* of data in its fixed-size communication FIFO for instance.

However, classical approaches with dataflow formalism make the assumption of an atomic execution of actors which is too restrictive when data transfers between actors on a real platform are concerned. Consider example of Fig. 8 with 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 bounds for the FIFO sizes). However, B could be triggered as soon as one token is produced on its input channel. Hence, if A is able to output one token at a time and if the platform provides the necessary synchronization facilities (basically blocking read/write operation on FIFOs), the size of the required FIFO can be limited to one.

Recent works addressed this problem of deriving tighter life-times for data, and thus smaller buffer sizes. M. Wiggers [2009] formalized a small-grained refinement of actors for parametric dataflow graphs. This work requires execution time of actors and uses a simulation-based approach, which is valid only in the cases where producer/-

consumer rates are known statically. Tong et al. [2012] applied similar techniques on radio applications with similar limitations.

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: the $set(p) \rightarrow get(p)$ dependency in schedule (5) forbids any firing of B before A has finished the production of all its 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 the next Section, we introduce *micro-schedules* as a way to explicitly express the relative dependencies between production and consumption of data and parameters.

5. MICRO-SCHEDULES

In this Section, we introduce our refinement to Fradet et. al. quasi-static scheduling formalism: the *micro-schedule* formalism for parametric data flow. Then we show how to use 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.

5.1. 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 call *micro-schedule*. It is important to note that micro-scheduling is not a new MoC, but a refinement of the SPDF semantics to enable more efficient schedules on real multi-core targets.

Micro-schedules express the sequential order of input and output operations 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, as it was the case for all SDR IP that we have used. The micro-schedule is extracted from actors' computation code for processors, or predefined for hardware accelerators IPs. One can also see micro-schedule as a granularity refinement and a generalisation of SDF to CSDF as it was done in [Geilen et al. 2011].

Formally, the micro-schedule for a SPDF graph includes the following instructions in addition to the components of quasi-static schedules introduced in Section 4.2:

— Actor A sending n tokens to actor B is denoted $push_{AB}(n)$
— Actor A receiving n tokens from actor B is denoted $pop_{AB}(n)$
— Actor testing for n^{th} execution during an iteration is denoted $it = n?$. As an actor micro-schedule may be executed repeatedly within a single schedule of the IPs (see schedule (6) below for instance), $it = n?inst$ will execute $inst$ only if the current micro-schedule instance is the n^{th} instance within one schedule.

Micro-schedule are expressed at actor level and we keep the term *schedule* for the schedule of the IP. All $push$ and pop instructions (i.e. data I/O) are expressed in actor schedule, however extra care is needed for parameter I/O that might occur in actor schedule or IP schedulexs. As we have seen in Section 4.2, 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 schedule of the IP on which it is mapped. The simplest case is an actor A which is fired only once per iteration: it produces a new parameter value at each execution, which makes the inclusion of the $set_A(p)$ inside the micro-schedule straightforward. But if the actor is fired several times in an iteration, the $it = n?$ test operator is mandatory to set which actor firing enables the production of the new parameter value.

Parameter Consumption. Parameters consumption are provided in IP schedule rather than in actor micro-schedule for two reasons. The first appears when an actor is scheduled a parametric number of times (e.g. $(get(p); A^p)$). In this case, getting the parameter is done in the IP schedule unambiguously. The second reason appears when an actor uses the parameter value to control 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 of the actor but given the fact that there is one refresh per iteration, it is safe to to keep the parameter's consumption (i.e. get) outside the actor's micro-schedule, i.e. in the IP schedule.

A valid micro-schedule for the actors of Fig. 8 is represented on the left part of schedule (6) below. Again, remember that finding a micro-schedule for each actor is out of the scope of this paper, most of the time it will be given in the actor or IP specification. Finding, for each actor, the best actor micro-schedule (e.g. to minimize FIFO buffer size globally) is a very complex problem to solve [Quinton and Risset 2001; Stuijk et al. 2011]. Actor schedules shown on the left of schedule (6) can be used for multi-core scheduling. With the previous mapping, the previous multi-core schedule (5) is changed to reflect the setting of parameters inside the IP schedules in the right part of schedule (6).

$$\begin{array}{ll} \mu S(A) = \left(set(p); \left(push_{AB}(1); push_{AC}(1)\right)^{2p}\right) & \\ \mu S(B) = \left(pop_{AB}(1); it = 1?set(q); push_{BC}(q)\right) & \\ \mu S(C) = \left(pop_{AC}(p); push_{CD}(p)\right) & S_{\text{IP}_1} = \left(\mu S(A); \mu S(C)^2\right) \\ \mu S(D) = \left(pop_{BD}(q); pop_{CD}(1)\right)^p & S_{\text{IP}_2} = \left(get(p); \mu S(B)^{2p}; \mu S(D)^2\right) \end{array} \tag{6}$$

With this schedule, the size of the FIFOs between A and B can be reduced to $|AB| = 1$ instead of $2p_{max}$ (i.e. B can be fired at each token produced by A). Similarly the size of the FIFO between C and D can be reduced to $|CD| = p_{max}$ instead of $2p_{max}$. Basically, micro-schedule does not change the scheduling, it allows the programmer to check more precisely if a given schedule and associated micro-schedule will deadlock or not for given sizes of FIFO between actors. In the example of schedule (6), a single 1-token large FIFO between A and B will not block the execution.

In the next Section we show how to solve efficiently, using this micro-schedule formalism, the following problem : given a multi-core quasi-static micro-schedule of a SPDF graph mapped on an architecture, are the FIFOs between the IPs of the architecture sufficiently large to avoid deadlock?

5.2. Checking Buffer Requirements

In the previous Section we introduced the concept of micro-schedule to describe actor behavior in a DFG. These micro-schedules lead to a deterministic, deadlock free execution, 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 the available buffer sizes. We want to check that, for *any* of real execution trace, no deadlock is reached, our approach is to walk through all possible execution traces thanks to the use of a model checker.

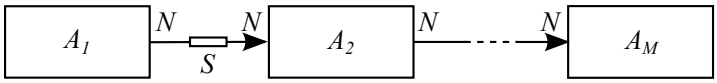


Fig. 9: Illustrating example of a static DFG mapped on an architecture with M IPs, one actor per IP. Each actor produces and/or consumes N tokens at each firing. FIFOs between IPs are of size S, and the target architecture allows each actor to consume and produce one token at a time.

Spin [Holzmann 2004] is an open-source model checker targeting verification of multi-threaded software. In particular, it has already been used for DFG scheduling [Geilen et al. 2005; Hartel and Ruys 2008; Liu et al. 2009; Malik and Gregg 2013]. In this work, we introduce a new model tailored for DFG verification to avoid state space explosion in the presence of concurrent actors. Verification results on LTE-Advanced examples are presented in Section 6.

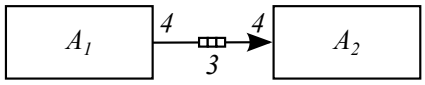
5.2.1. A new modeling technique for buffer size verification. To illustrate the technique, let us consider the generic application presented on Fig. 9: a simple chain of M actors, each executing on one IP. Actors run concurrently and communicate through channels of size S. The behavior of each actor is given by its micro-schedule:

— Source actor A_1: $\mu S(A_1) = \left(push_{A_1,A_2}(1)^N\right)$;
— Actor A_x, $1 < x < M$: $\mu S(A_x) = \left((pop_{A_{x-1},A_x}(1); push_{A_x,A_{x+1}}(1))^N\right)$;
— Sink actor A_M: $\mu S(A_M) = \left(pop_{A_{M-1},A_M}(1)^N\right)$.

In order to illustrate the promela code (Fig. 10b) we use a particular instance of this application, depicted on Fig. 10a and characterized by 2 actors ($M = 2$), 4 tokens exchanged ($N = 4$) and a FIFO of size 3 ($S = 3$). We want to verify that the above schedule will never end in a deadlock at execution time. For that we can use SPIN [Holzmann 2004] and the Promela modeling described in [Dardaillon et al. 2014a], that we will call *global memory model*, or we can use the modeling that we propose hereafter and that we call *channel memory model*. The main difference between the global memory model and the channel memory model is that the FIFOs are modeled by the **`chan`** Promela primitive in the channel memory model while they are modeled by a simple integer in the global memory model. Let us consider first the behavior of the two models in broad terms on the example of Fig. 10a, before formally defining the channel memory model.

Global memory model. In the first step of the execution actor A_1 produces 1 token. From this point, actor A_1 can produce another token followed by actor A_2 consuming a token, or actor A_2 can consume one token before actor A_1 produces another token. These two possibilities are two different execution traces, although they both lead to the same final state. Previous works [Geilen et al. 2005; Hartel and Ruys 2008; Liu et al. 2009; Malik and Gregg 2013; Dardaillon et al. 2014a] on dataflow scheduling use global variables to model FIFOs sizes. This method links all actors to the global state in SPIN, although each actor is only dependent on its input and output FIFOs. Hence, all possible traces are explored, which leads to a state space explosion.

Channel memory model. In the example actors A_1 and A_2 are executed in parallel, and they have an exclusive read/write access to a single FIFO, which means that their *partial execution order* has no influence on the execution result. The partial order reduction technique [Holzmann and Peled 1994] exploits the commutativity of concurrently executed transitions to reduce the state space. In this work, we propose to use the channel primitive of Promela to model each FIFO. Using this primitive, the SPIN



(a) Mapped dataflow graph.

```
chan ch_1 = [3] of {bit}

active proctype A_1() {
  xs ch_1;
  byte i;
  for(i:1 .. 4) {
    ch_1!0;
  }
}
```

```
active proctype A_2() {
  xr ch_1;
  byte i;
  for(i:1 .. 4) {
    ch_1?0;
  }
}
```

(b) Proposed channel memory Promela model.

Fig. 10: Instance of mapped dataflow graph from Fig. 9 ($M = 2, N = 4, S = 3$) and associated Promela code.

model checker is able to apply partial order reduction, resulting in dramatic improvement in the number of states explored.

We now define the channel memory model using Promela, and model the example of Fig. 10a in Promela code on Fig. 10b:

— Each core is encoded as a Promela process **proctype**. To cope with the micro-schedule paradigm, we are refining the dataflow MoC by removing the atomic actor execution hypothesis. All the processes are marked as **active**, which means that they are all running concurrently at start time.
— The FIFOs (i.e. blocking read and write FIFOs) are modeled using the Promela channel **chan** `ch_x` primitive. For example: **chan** `ch_1 = [3]` **of** {**bit**} represents a FIFO of size 3. The writing (resp. reading) of a single token belonging to arc `y` and mapped to FIFO `ch_x` is modeled by `ch_x!y` (resp. `ch_x?y`). The **xs** (resp. **xr**) primitive signals that the process is the only producing (resp. consuming) tokens on the FIFO. This **chan** primitive is essential for partial order reduction in order to reduce the state space. In the global memory model [Dardaillon et al. 2014a], FIFOs are modeled by simple global integer variables.
— Parameters are modeled similarly using the Promela **chan** `p_x` primitive. The production (resp. consumption) of a parameter `p` with value `y` is modeled by `p_x!y` (resp. `p_x?y`). Note that SPIN is not a symbolic model checker and that therefore, all possible values of the parameters are explored by SPIN using the **select**`(p:1..pMAX)` primitive.

Once the Promela specification is written as in Fig. 10b, SPIN attempts to verify that all execution traces lead to a correct end state, i.e. that all processes have ended their execution and all FIFOs are empty. If initial tokens need to be set on some edges, additionnal constraints can be added to assert that the correct number of tokens are left in the edges.

5.2.2. SPIN *models performance comparison.* To evaluate the performances of the two models for buffer verification we use the application of Fig. 9. Loop unrolling was applied to all global memory models to reduce the number of states of each actor. Results are presented on Fig. 11, measuring the number of states stored by SPIN for a variable number of data exchanged ($N \in [1:10]$), with a variable number of actors ($M \in [2:6]$)

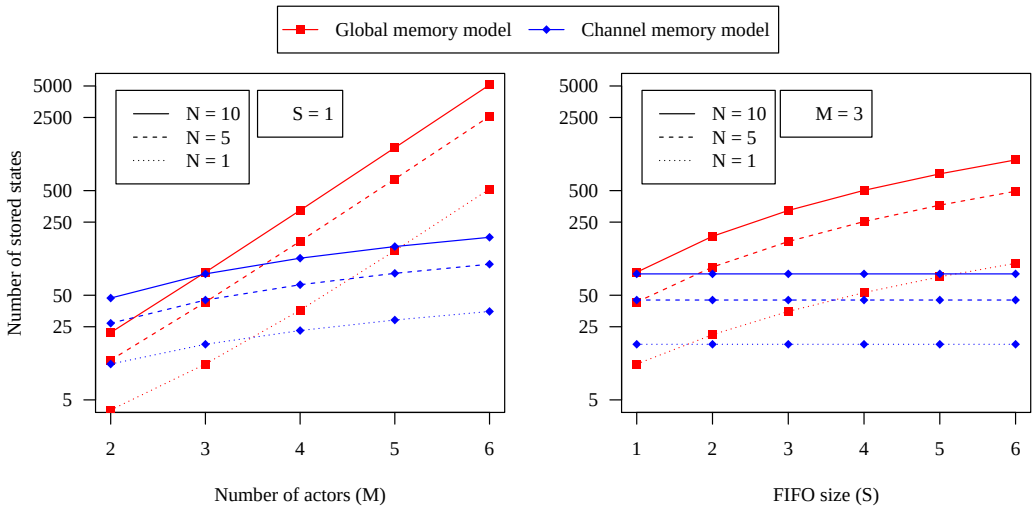


Fig. 11: Number of states stored by SPIN for the generic application from Fig. 9 with various values of N, M and S, using global and channel memory models: the use of the channel memory model greatly decreases the complexity of the resolution.

on the left and a variable FIFO size ($S \in [1:6]$) on the right. The scale of the number of state is logarithmic

The most remarkable improvement of the channel memory model can be seen on the left projection: the growth of the number of states is linear for the channel model (although it is not obvious with the logarithmic scale), while the growth is exponential for the global memory model. As mentionned before, this can be explained because in the channel model, each process modeling an actor is independent of other processes, which results in a total number of states proportional to the sum of states of each process. On the other hand in the global memory model all processes modeling actors are linked by the global variables modeling their FIFOs, which results in a total number of states proportionnal to the product of the number of state of each process. The result is a state explosion even for very small numbers of actors in our example.

Illustrated in the right projection is the influence of the FIFO size. Increasing the size of one FIFO results in a larger number of possible executions, which in turn increases the number of states of the global memory model. In addition to these results, we provide complexity results for the two Promela models applied to parts of LTE procotol in Section 6.

In summary, we have shown that the association of micro-schedule with model checking offers a tool to control very precisely the pipeline between two IPs. In each architecture using hardware FIFOs, the sizes of these FIFOs are small (because inter-IP hardware FIFOs are very costly), and these sizes can be fixed. In Section 6, we show how we have used this technique to check for the first time the absence of deadlock on the Magali platform.

6. EXPERIMENTAL RESULTS

In this Section we analyze the performance of our compiler: performance of the code development, performance of the code generated and performance of the buffer verification technique.

Table I: Comparison of compiled code and handwritten code targeting Magali.

	PaDaF		Native	
Application	C++ (#lines)	(hours)	C / ASM (#lines)	(weeks)
OFDM	60	1	150 / 200	1
Demodulation	160	4	300 / 600	4
Parametric Demod.	260	8	500 / 800	12

Table II: Buffer checking results on the SPIN model checker.

	Global memory model		Channel memory model	
Application	stored states	run time (s)	stored states	run time (s)
OFDM	12798	< 0.1	5771	< 0.1
Demodulation	1.99e7	54	4986	< 0.1
Full Demod.	5.54e7	81	5360	< 0.1
Parametric Demod.	6.06e7	121	11574	< 0.1

Code development performance. The benefits of using our compiler are described in Tab. I. The estimation of the time for writing native code to Magali is not based on our experience but on the experience of engineers that programmed the LTE-Advanced on Magali. As mentioned before, this very long manual programming process was the main motivation for the development of a compiler for Magali.

Of course, required time to write an application is a subjective metric, because its process includes reflection times which are 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 estimation 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.

Buffer Checking Technique performance. 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. II. These simulations were run on a 2.8 GHz Intel Core i5 with 8 GB of RAM running OS X 10.10.2, with SPIN 6.4.2 and GCC 4.9.2. Promela models of the different test cases were generated as described in Section 5.2.1. An additional *full demodulation* test case based on *parametric demodulation* with only the largest parameter value was added to evaluate the influence of the parameter variation on the verification.

The results demonstrate again the strength of the channel memory model to verify applications involving a large number of actors. 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 caused by communications for these applications.

Table III: Comparison between SPIN and SDF^3 on dynamic applications.

	SPIN channel memory		SDF^3 [Theelen et al. 2011]	
Application	stored states	run time (s)	stored states	run time (s)
MPEG-4 AVC	184	< 0.1	22	< 0.1
Channel Equalizer	340	< 0.1	297	< 0.1
MPEG-4 SP (PD = 1)	5185	< 0.1	38441	2
MP3 (PD = 1)	710404	0.66	-	-
OFDM	5771	< 0.1	15756	1
Demodulation	4986	< 0.1	12002	5
Full Demod.	5360	< 0.1	6177	1
Parametric Demod.	11574	< 0.1	52581	215

Table IV: Performance result of generated code with respect to handwritten code

Application	handwritten (μs)	[Risset et al. 2011] (μs)	generated (μs)	optimized (μs)
OFDM	149	500 (+236%)	168 (+13%)	149 (+0%)
Demodulation	180	-	283 (+57%)	180 (+0%)
Parametric Demod.	419	-	558 (+33%)	288 (-31%)

SDF^3 [Stuijk et al. 2006] is a reference for dataflow analysis, with tests such as consistency and throughput computation for SDF, CSDF and SADF graphs. SADF in particular is more expressive than SPDF, and has already been used to model the LTE-Advanced application [Siyoum et al. 2011]. Differences between SDF^3, which is more focused on throughput, and our model, which focus only on deadlock detection, make the direct comparison of the two methods difficult. With this fair warning, we present results from both methods in Table III.

Dynamic applications from [Theelen et al. 2011] were ported to our Promela channel memory model using the SPDF MoC, extended (enabling parameters taking zero value) to match the SADF expressivity. We supposed a 1 to 1 mapping between actors and cores, and buffer size of 2 tokens, same size as on Magali platform. LTE-Advanced applications were also ported to SDF^3 using the platform mapping, buffer sizes and micro-scheduling used in the channel memory model. SDF^3 represents cyclic production and consumption of data as series of states in the SADF model, which does not fare well with the idea of micro-scheduling. As such, each actor with a micro-schedule is encoded using a dedicated SADF detector to define its current production and consumption rates, increasing the complexity of the model.

All experiments using SPIN and SDF^3 were run on the same test machine to have comparable run time. SPIN was able to verify the absence of deadlock for all applications, including complex dynamic applications such as MP3, in less than a second. The same applications were analyzed for their reduced state space (SDF^3 was run to analyse the number of states after solving non determinism [Theelen et al. 2011]) using SDF^3. From these results we observe a larger number of stored states and run time due to the different type of analysis ran by SDF^3, and a failure to analyze the MP3 application. Although these results are for different analyses, they assess the performance of the proposed buffer checking method.

Performance of the code generated. The performance for the applications described in Section 2.2 are presented in Table IV. The *handwritten* code, used as a baseline to compare our solutions, is a porting of the 3GPP LTE-Advanced application previously explained [Clermidy et al. 2009b].

The code generated by our compilation flow is denoted as *generated*. The *optimized* code is the same code with manual optimizations on the central controller, described in the following. These optimizations are fully automatisable.

The overhead of the *generated* approach, compared to manual code, vary from 13% for small applications up to 57%, and is due to the central controller latency. To understand this latency, you have to look closer at the Magali control mechanisms [Clermidy et al. 2009a]. Each of the heterogeneous core running the application embeds a dedicated controller, which ability is limited to executing a sequence of configurations. The ARM processor used as a central controller is in charge of reconfiguring the distributed controllers based on the configurations to run and potential parameters value influence.

The *handwritten* approach splits dynamic applications in static phases, with global synchronization and reconfiguration between each phase being carried out by a single thread on the ARM processor. In the *generated* approach, each distributed core is controlled by a dedicated thread on the ARM processor, with the objective of pipelining the reconfigurations of the different cores. However, the reconfiguration time of each core is larger than the potential pipelining. This reconfiguration time, combined with interruptions from each core requesting a new configuration sequence, results in an overall higher latency. The *optimized* approach uses a single control thread on the ARM processor, which only reconfigure the cores depending on parameters.

This *optimized* approach removes a large part of the reconfiguration latency, and even improves the performance in the parametric application by reducing the number of reconfigurations compared to the *handwritten* approach. This optimization was done manually by modifying C code, and should be automated in the future. As the compiler knows which actors are dependant on which parameters as well as the mapping of actors onto hardware cores, this automation can be automated.

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.

7. 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 [Gonzalez et al. 2009] (implementing SCA), SPEX [Lin et al. 2006] or DiplodocusDF [Gonzalez-Pina et al. 2012] (see [Dardaillon et al. 2013] for a complete survey).

Up to now, few SDR programming environments have been adapted to more than one hardware architecture. GNUradio is adapted to low-performance radio applications but cannot address demanding applications such as LTE-Advanced in real-time. PREESM [Pelcat et al. 2014; Heulot et al. 2014] proposes a compilation flow for heterogeneous multicore DSPs, while we address a more complex heterogeneous platform with both DSP and accelerators. PREESM allows developers to use parameters as compile time constants to construct the graph, and hence to optimize generic components at compilation time. Our compilation flow provides a similar optimization, with use of constant parameters during graph construction and constant parameter propagation during analysis. PREESM proposes to use JIT scheduling to manage runtime parameters, which is not adapted to the Magali platform constraints [Risset et al. 2011]. MAPS [Castrillon et al. 2011] may be the compilation flow closest to ours. In particular, it addresses the compilation of telecommunications applications on heterogeneous platforms. Their approach of platform independent API (*nuclei*) and library of opti-

mized implementation (*flavors*) indeed inspired our work. However MAPS uses Khan process network (KPN) formalism in which deadlock detection is undecidable.

Many SDR programming environments are adopting the dataflow MoC. Some MoCs hold much information, offering various levels of static verification and optimization, such as SDF. Others allow very dynamic behaviors, such as KPN, see [Johnston et al. 2004; Dardaillon et al. 2013] for recent surveys. Recently, the need for verifiable but still flexible dataflow MoCs lead to the appearance of two new kinds of dataflow MoCs: Scenario-Aware DataFlow (SADF) [Stuijk et al. 2011] and parametric dataflow [Fradet et al. 2012; Bhattacharya and Bhattacharyya 2001]. MCDF, a sibling to SADF for both analysis and compilation, has already been used to implement the LTE-Advanced application successfully [Salunkhe et al. 2014]. In this work we chose to look at Fradet et al. subclass of parametric dataflow, i.e. Schedulable Parametric Dataflow (SPDF), where the schedulability of the DFG can still be assessed statically. This model is well adapted to our constraints as it provides enough expressivity for describing modern wireless waveforms, while allowing static analysis of buffer constraints, and quasi-static schedule needed for efficient code generation on Magali.

Concerning input language issues, one way to include complex graph construction is to rely on template, or macro, to describe the graph and use the preprocessor to generate it at compile time, such as in Ptolemy classic [Buck et al. 1994]. The main limitation of this approach is the expressivity of the template language. To cope with this problem, our flow uses C++ programming language and executes it at compilation time. The only other dataflow language providing such complex graph construction we are aware of is ΣC [Goubier et al. 2011], which uses a CSDF MoC without parameters and targets the MPPA homogeneous manycore platform. The complex dataflow graph construction is handled by a new language and compilation flow, while we propose to use an existing language and front-end (LLVM) to construct the graph. Our solution relies on a strong software tool community and provides a simpler environment for complex dataflow graph construction. ΣC and LIME [Kourzanov et al. 2010] use array to specify input and output data access in the actor, allowing the back-end to generate double-buffering or in-place data manipulation based on the platform. It requires all data to be available at the beginning of the actor execution which would not fit Magali constraints, but could be considered for other more flexible platforms.

Scheduling for buffer minimization is NP-complete [Bhattacharyya et al. 1999]. Many heuristics have been developed to schedule under memory constraints [Karczmarek et al. 2003; Geilen et al. 2005; Bhattacharyya et al. 1999]. We focussed on model-checking solutions based on the work by Geilen et al. [2005], which solves the scheduling problem on constrained buffer size for synchronous DFG. Using a similar approach, Damavandpeyma et al. [2012] minimize buffer on scheduled synchronous DFG. SDF^3 is a reference in dataflow analysis and we compared our work with theirs in Section 6. Our work concentrates on a subproblem of the buffer minimization, namely the absence of deadlock, as buffer sizes are already constrained by the platform. Ghamarian et al. [2006] proposed to check liveness of a dataflow graph using symbolic execution for SDF. Model checking techniques can be seen as way to explore this symbolic execution to prove the absence of deadlock. Several work propose to prove liveness for CSDF [Benazouz et al. 2013] and BPDF [Bebelis et al. 2013a] without symbolic execution. However they both propose an upper bound on the minimum buffer size which can be an issue for platform with fixed size buffers. In this context, the originality of our work is twofold. The modelization of *parametric* DFG reduces the complexity of the verification compared to verifying every possible SDF. The use of a finer grain modelization with micro-schedule enable checking the absence of deadlock on scheduled DFGs with strong memory constraints.

The work of Wiggers [2009] and Tong et al. [2012] formalized refinements of actors close to our micro-schedules, but only applicable to CSDF. They use their model of actors to compute minimal buffer sizes, using a simulator and actors' execution times.

8. CONCLUSION

This paper presents a new compilation flow, based on the LLVM framework, that compiles parametric dataflow graphs down to heterogeneous MPSoC. This framework is dedicated to new wireless applications using new models of computation such as parametric dataflow appearing for signal processing applications. We also introduce a format based on C++ to express complex parametric graphs as well as the micro-schedule formalism to describe actors communication behavior in dataflow graphs. Based on this formalism, we provide a new buffer size verification method using model checking which can be performed when mapping dataflow graph on MPSoCs.

To validate our results, experiments on the Magali platform are performed using parts of the LTE-Advanced protocol. All test cases are successfully checked for their buffer usage with a significant improvement on the verification time thanks to the new verification method. The performances of the programs generated by our compiler are very close to the corresponding handwritten programs: the maximum overhead observed is 57% and in some cases the compiled programs are even more efficient than the corresponding handwritten programs which are, of course, much more time consuming to produce.

REFERENCES

2015. CLANG: a C language family frontend for LLVM. (Feb. 2015). http://clang.llvm.org

2015. The LLVM Compiler Infrastructure. (Feb. 2015). http://llvm.org

Vagelis Bebelis, Pascal Fradet, Alain Girault, and Bruno Lavigueur. 2013a. A Framework to Schedule Parametric Dataflow Applications on Many-Core Platforms. In *17th workshop on Compilers for Parallel Computing, CPC*. Lyon, FR.

Vagelis Bebelis, Pascal Fradet, Alain Girault, and Bruno Lavigueur. 2013b. BPDF: A statically analyzable dataflow model with integer and boolean parameters. In *Proc. International Conference on Embedded Software (EMSOFT)*. Montreal, QC, 1–10.

Mohamed Benazouz, Alix Munier-Kordon, Thomas Hujsa, and Bruno Bodin. 2013. Liveness evaluation of a cyclo-static DataFlow graph. In *Proceedings of the 50th Annual Design Automation Conference (DAC)*. ACM, Austin, Texas, 3.

Heikki Berg, Claudio Brunelli, and Ulf Lucking. 2008. Analyzing models of computation for software defined radio applications. In *Proc. International Symposium on System-on-Chip*. Tampere, Finland, 1–4.

B. Bhattacharya and S.S. Bhattacharyya. 2001. Parameterized Dataflow Modeling for DSP Systems. *Trans. Sig. Proc.* 49, 10 (Oct. 2001), 2408–2421.

Shuvra S. Bhattacharyya, Praveen K. Murthy, and Edward A. Lee. 1999. Synthesis of embedded software from synchronous dataflow specifications. *Journal of VLSI signal processing systems for signal, image and video technology* 166, 2 (1999), 151–166.

Joseph T Buck, Soonhoi Ha, Edward A Lee, and David G Messerschmitt. 1994. Ptolemy: A framework for simulating and prototyping heterogeneous systems. *International Journal of Computer Simulation, special issue on "Simulation Software Development"* 4 (April 1994), 155–182.

João M. P. Cardoso, Pedro C. Diniz, and Markus Weinhardt. 2010. Compiling for reconfigurable computing. *Comput. Surveys* 42, 4 (June 2010), 1–65.

Jeronimo Castrillon, Stefan Schürmans, Anastasia Stulova, Weihua Sheng, Torsten Kempf, Rainer Leupers, Gerd Ascheid, and Heinrich Meyr. 2011. Component-based waveform development: The Nucleus tool flow for efficient and portable software defined radio. *Analog Integrated Circuits and Signal Processing* 69, 2-3 (June 2011), 173–190.

Fabien Clermidy, Christian Bernard, Romain Lemaire, Jerome Martin, Ivan Miro-Panades, Yvain Thonnart, Pascal Vivet, and Norbert Wehn. 2010. A 477mW NoC-based digital baseband for MIMO 4G SDR. In *Proc. IEEE International Solid-State Circuits Conference, ISSCC*. San Francisco, CA, 278–279.

Fabien Clermidy, Romain Lemaire, Xavier Popon, Dimitri Ktenas, and Yvain Thonnart. 2009b. An Open and Reconfigurable Platform for 4G Telecommunication: Concepts and Application. In *12th Euromicro Conference on Digital System Design, Architectures, Methods and Tools*. Patras, Greece, 449–456.

Fabien Clermidy, Romain Lemaire, and Yvain Thonnart. 2009a. A Communication and configuration controller for NoC based reconfigurable data flow architecture. In *3rd ACM/IEEE International Symposium on Networks-on-Chip (NOCS)*. San Diego, California, 153–162.

M. Damavandpeyma, Sander Stuijk, Twan Basten, Marc Geilen, and Henk Corporaal. 2012. Modeling static-order schedules in synchronous dataflow graphs. In *2012 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. Dresden, Germany, 775–780.

Mickaël Dardaillon, Kevin Marquet, Jérôme Martin, Tanguy Risset, and Henri-Pierre Charles. 2013. *Cognitive Radio Programming: Existing Solutions and Open Issues*. Technical Report September. Inria.

Mickaël Dardaillon, Kevin Marquet, Tanguy Risset, Jérôme Martin, and Henri-Pierre Charles. 2014a. A Compilation Flow for Parametric Dataflow: Programming Model, Scheduling, and Application to Heterogeneous MPSoC. In *International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*. New Delhi, India.

Mickaël Dardaillon, Kevin Marquet, Tanguy Risset, Jérôme Martin, and Henri-pierre Charles. 2014b. 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.

Pascal Fradet, Alain Girault, and Peter Poplavko. 2012. SPDF: A schedulable parametric data-flow MoC. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. Dresden, Germany, 769–774.

Marc Geilen, Twan Basten, and Sander Stuijk. 2005. Minimising buffer requirements of synchronous dataflow graphs with model checking. In *Conference on Design Automation (DAC)*. San Diego, CA, 819.

Marc Geilen, Stavros Tripakis, and Maarten Wiggers. 2011. The earlier the better: a theory of timed actor interfaces. In *International conference on Hybrid systems: computation and control (HSCC)*. 23–32.

Amir Hossein Ghamarian, Marc Geilen, Twan Basten, Bart D Theelen, Mohammad Reza Mousavi, and Sander Stuijk. 2006. Liveness and boundedness of synchronous data flow graphs. In *Formal Methods in Computer Aided Design (FMCAD)*. IEEE, San Jose, California, 68–75.

C.R.A. Gonzalez, C.B. Dietrich, Shereef Sayed, H.I. Volos, J.D. Gaeddert, P.M. Robert, J.H. Reed, and F.E. Kragh. 2009. Open-source SCA-based core framework and rapid development tools enable software-defined radio education and research. *IEEE Communications Magazine* 47, 10 (Oct. 2009), 48–55.

Jair Gonzalez-Pina, Rabea Ameur-Boulifa, and Renaud Pacalet. 2012. DiplodocusDF, a Domain-Specific Modelling Language for Software Defined Radio Applications. In *38th Euromicro Conference on Software Engineering and Advanced Applications*. Cesme, Izmir, 1–8.

Thierry Goubier, Renaud Sirdey, Stéphane Louise, and Vincent David. 2011. ΣC A Programming Model and Language for Embedded Manycores. In *Algorithms and Architectures for Parallel Processing - 11th International Conference, ICA3PP*. Melbourne, Australia, 385–394.

Pieter H. Hartel and Theo C. Ruys. 2008. Scheduling optimisations for SPIN to minimise buffer requirements in synchronous data flow. In *International Conference on Formal Methods in Computer-Aided Design (FMCAD)*. Portland, Oregon.

Julien Heulot, Maxime Pelcat, Jean-François Nezan, Yaset Oliva, Slaheddine Aridhi, and Shuvra S Bhattacharyya. 2014. Just-in-time scheduling techniques for multicore signal processing systems. In *Global Conference on Signal and Information Processing (GlobalSIP)*. IEEE, Atlanta, Georgia, 25–29.

Gerard J Holzmann. 2004. *The SPIN model checker: Primer and reference manual*. Vol. 1003. Addison-Wesley Reading.

Gerard J Holzmann and Doron Peled. 1994. An improvement in formal verification.. In *International Conference on Formal Description Techniques (FORTE)*. Bern, Switzerland, 197–211.

Wesley M. Johnston, J. R. Paul Hanna, and Richard J. Millar. 2004. Advances in dataflow programming languages. *Comput. Surveys* 36, 1 (March 2004), 1–34.

Shin-Haeng Kang, Hoeseok Yang, L. Schor, I. Bacivarov, Soonhoi Ha, and L. Thiele. 2012. Multi-objective mapping optimization via problem decomposition for many-core systems. In *Embedded Systems for Real-time Multimedia (ESTIMedia), 2012 IEEE 10th Symposium on*. 28–37.

Michal Karczmarek, William Thies, and Saman Amarasinghe. 2003. Phased scheduling of stream programs. In *Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*. San Diego, CA, 103.

Pjotr Kourzanov, Orlando Moreira, and Henk J Sips. 2010. Disciplined Multi-core Programming in C.. In *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*. Las Vegas, Nevada, 346–354.

Seongnam Kwon, Yongjoo Kim, Woo C. Jeun, Soonhoi Ha, and Yunheung Paek. 2008. A retargetable parallel-programming framework for MPSoC. *ACM Trans. Des. Autom. Electron. Syst.* 13, 3 (2008), 1–18.

Edward A. Lee and David G. Messerschmitt. 1987. Synchronous data flow. *Proc. IEEE* 75, 9 (June 1987), 1235–1245.

Yuan Lin, Robert Mullenix, Mark Woh, Scott Mahlke, Trevor Mudge, Alastair Reid, and Krisztian Flautner. 2006. SPEX: A programming language for software defined radio. In *SDR Forum Technical Conference*. Orlando, Florida, 13 – 17.

Weichen Liu, Zonghua Gu, Jiang Xu, Yu Wang, and Mingxuan Yuan. 2009. An efficient technique for analysis of minimal buffer requirements of synchronous dataflow graphs with model checking. In *International conference on Hardware/software codesign and system synthesis (CODES+ISSS)*. 61–70.

Avinash Malik and David Gregg. 2013. Orchestrating stream graphs using model checking. *ACM Transactions on Architecture and Code Optimization (TACO)* 10, 3 (Sept. 2013), 19.

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

Preeti Ranjan Panda. 2001. SystemC - A modeling platform supporting multiple design. In *Proc. 14th International Symposium on Systems Synthesis (ISSS)*. Montreal, QC, 75–80.

Maxime Pelcat, Karol Desnos, Julien Heulot, Clément Guy, Jean-François Nezan, and Slaheddine Aridhi. 2014. PREESM: A Dataflow-Based Rapid Prototyping Framework for Simplifying Multicore DSP Programming. In *European Embedded Design in Education and Research Conference (EDERC)*. 36–40.

P. Quinton and T. Risset. 2001. Structured Scheduling of Recurrence Equations: Theory and Practice. In *Proc. of the System Architecture MOdelling and Simulation Workshop (Lecture Notes in Computer Science, 2268)*. Springer Verlag, Samos, Greece, 112–134.

Tanguy Risset, Riadh Ben Abdallah, Antoine Fraboulet, and Jérôme Martin. 2011. *Digital Front-End in Wireless Communications and Broadcasting*. Cambridge University Press, Chapter Programming models and implementation platforms for software defined radio configuration, 650–670.

Hrishikesh Salunkhe, Orlando Moreira, and Kees van Berkel. 2014. Mode-controlled dataflow based modeling & analysis of a 4g-lte receiver. In *Proceedings of the conference on Design, Automation & Test in Europe (DATE)*. 212.

Firew Siyoum, Marc Geilen, Orlando Moreira, Rick Nas, and Henk Corporaal. 2011. Analyzing synchronous dataflow scenarios for dynamic software-defined radio applications. In *International Symposium on System on Chip (SoC)*. IEEE, Tampere, Finland, 14–21.

Sander Stuijk, Marc Geilen, and Twan Basten. 2006. SDF3: SDF For Free. In *International Conference on Application of Concurrency to System Design (ACSD)*. IEEE, Turku, Finland, 276–278.

Sander Stuijk, Marc Geilen, Bart Theelen, and Twan Basten. 2011. Scenario-aware dataflow: Modeling, analysis and implementation of dynamic applications. In *International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation*. Samos, Greece, 404–411.

Bart Theelen, Marc Geilen, and Jeroen Voeten. 2011. Performance model checking scenario-aware dataflow. In *Proceedings of the International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS)*, LNCS (Ed.), Vol. 6919. Springer, 43–59.

Wei Tong, O. Moreira, R. Nas, and K. van Berkel. 2012. Hard-Real-Time Scheduling on a Weakly Programmable Multi-core Processor with Application to Multi-standard Channel Decoding. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2012 IEEE 18th*. 151–160.

Maarten Hendrik Wiggers. 2009. *Aperiodic multiprocessor scheduling for real-time stream processing applications*. Ph.D. Dissertation. Enschede.

Mark Woh, Yoav Harel, Scott Mahlke, Trevor Mudge, Chaitali Chakrabarti, and Krisztian Flautner. 2006. SODA: A Low-power Architecture For Software Radio. In *33rd International Symposium on Computer Architecture, ISCA*. Boston, MA, 89–101.

Mark Woh, Sangwon Seo, Hyunseok Lee, Yuan Lin, Scott Mahlke, Trevor Mudge, Chaitali Chakrabarti, and Krisztian Flautner. 2007. The next generation challenge for software defined radio. In *Embedded Computer Systems: Architectures, Modeling, and Simulation*. Samos, Greece, 343–354.

Jim Zyren and W. McCoy. 2007. *Overview of the 3GPP long term evolution physical layer*. Technical Report. Freescale Semiconductor Inc.

Received July 2015; revised January 2016; accepted NA