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

MICKAËL DARDAILLON, KEVIN MARQUET, and TANGUY RISSET, Université de Lyon, INRIA, INSA-Lyon, CITI-INRIA

JÉRÔME MARTIN, Université Grenoble Alpes, CEA, LETI, Minatec Campus

HENRI-PIERRE CHARLES, Université 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 behavior 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 article 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 that currently matches the Magali platform: a prototype heterogeneous MPSoC dedicated to LTE-Advanced. We also introduce an innovative scheduling technique, called *microscheduling*, allowing one 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 performance similar to handwritten code.

CCS Concepts: • **Software and its engineering** → **Data flow languages**; **Retargetable compilers**

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. 2016. A new compilation flow for software-defined radio applications on heterogeneous MPSoCs. *ACM Trans. Archit. Code Optim.* 13, 2, Article 19 (June 2016), 25 pages.

DOI: <http://dx.doi.org/10.1145/2910583>

This work was sponsored by Région Rhône Alpes ADR 11 01302401.

Part of this work was published as M. Dardaillon, K. Marquet, T. Risset, J. Martin, and H.-P. Charles. 2014. A compilation flow for parametric dataflow: Programming model, scheduling, and application to heterogeneous MPSoC." In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES'14)*. This work extends the previous publication by providing a thorough presentation of a new front end technique, introducing a new promela model for buffer size verification, and updated results reflecting the substantial improvements provided by the new promela model.

Authors' addresses: M. Dardaillon, National Instruments Corporation, 11500 N Mopac Expy, 78759 Austin, Texas; email: mickael.dardaillon@ni.com; T. Risset and K. Marquet, CITI, 6 avenue des arts, 69621 Villeurbanne Cedex, France; emails: {tanguy.risset, kevin.marquet}@insa-lyon.fr; J. Martin and H.-P. Charles, CEA, Minatec Campus, 17 rue des martyrs, 38054 Grenoble Cedex, France; emails: {jerome.martin, henri-pierre.charles}@cea.fr.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.

© 2016 ACM 1544-3566/2016/06-ART19 \$15.00

DOI: <http://dx.doi.org/10.1145/2910583>

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, and video decoding. 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 that are not possible with today’s wireless devices. These technological trends have reactivated 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 recently led to the appearance of new parametric dataflow models of computation (MoCs).

A typical example to illustrate dataflow dynamicity comes from LTE-Advanced: the type of modulation (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 microseconds. 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 MoC.

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 500mW). 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 toward a more generic approach: compiling SDR waveforms from high-level dataflow representations rather than manual tuning.

The contributions provided by this work are as follows:

- A new compilation flow for 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, analyzes, and generates an internal representation for the parameterized dataflow graph (DFG) described in C++, as well as a back end dedicated to the Magali MPSoC.
- A new high-level format for expressing parametric DFGs in reduced form. This format permits one to express DFGs 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 *microschedule* that permits a more precise analysis of deadlock when mapping a parametrized DFG to real architecture. We also present improved model checking use to the specific problem of

advanced actor pipelining in the context of a dedicated target architecture with small buffers.

The article is organized as follows. Section 2 presents the context of the work, Magali target architecture, and specificities of the modern wireless waveform that motivates the development of a new programming paradigm. Section 3 presents our compilation framework. Section 4 presents parametric dataflow scheduling and shows why the existing scheduling techniques are not adapted to the Magali target. We introduce microscheduling 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 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. Earlier, we mentioned 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 SDR 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.

SDR technology uses a wide variety of execution models: homogeneous and heterogeneous multicores such as commercial baseband processors from companies like Texas Instruments, Qualcomm, or Freescale; FPGA-based machines; dedicated ASIC; cloud-RAN architectures; among others. 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 (i.e., LTE-Advanced) 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, and so forth. 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 the waveform applications.

2.1. Magali Hardware Architecture

The Magali chip [Clermidy et al. 2009b] represented in Figure 1 is an SoC dedicated to physical layer processing of OFDMA radio protocols, with a special focus on 3GPP LTE-Advanced as a 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 done by the CPU in the case of complex applications. These distributed control sequences,

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

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

called *contexts*, were 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 motivations to write a compiler for Magali.

As 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 that indeed correspond to difficult problems for designers to solve. The first problem comes from the complexity of the protocols: they include many blocks in parallel as illustrated in Figure 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 particularly includes 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 adaptive 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 of these reconfigurations

have to be done within approximately 1ms. This fast dynamic reconfiguration motivates the building of a dedicated chip such as Magali and the use of a parameterized dataflow 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

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] and Clermidy et al. [2009b]. The implemented test case applications correspond to *ofdma* and *channel decoder* of LTE-Advanced. The test case applications are represented in Figure 2(a) through (c) and are described hereafter.

OFDM test case. The OFDM test case, presented in Figure 2(a), 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 mentioned in Section 2.1.

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

Parametric demodulation test case. Parametric demodulation (Figure 2(c)) 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 simultaneously describe 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 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 Figure 3, is split into two phases: a front end for parsing and analysis of the DFG is introduced in Section 3.2, and a 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.

3.1. Parametric Dataflow Format: *padaf*

The input format follows the schedulable parametric dataflow (SPDF) MoC proposed by Fradet et al. [2012]. The SPDF graph in Figure 4 illustrates a MIMO receiver with

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

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

four antennas. The set `p[1]` in actor `Decod` indicates that actor `Decod` produces a new integer value for parameter `p` each time it fires. In addition, the graph indicates that actor `Decod` defines `p` and then outputs `57p` tokens (the reader should refer to Fradet et al. [2012] for a detailed presentation of SPDF).

The input format that we propose—parametric dataflow format (PaDaF)—allows us to describe both actor behavior in C++ and a parametric DFG using the actors. Figure 5(a) and (b) present an actor 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 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 the `Ofdm` class. However, specifying that number can be more complex because this number might be parametric (see the output port of the `Decod` actor), and as the number of ports of an actor might be symbolic (e.g., `nbAnt` input ports for the `Decod` actor), we might need some code to specify the number of tokens on each

<pre> class Ofdm : public Actor { PortIn<int> Iin; PortOut<int> Iout; void compute(); Ofdm(): Iin(1024), Iout(600){} }; </pre>	<pre> 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); } }; </pre>
(a) Actor OFDM declaration	(b) Actor Decod declaration

Fig. 5. Actor OFDM and Decod declarations in PaDaF.

<pre> 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; </pre>	<pre> 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); } </pre>
(a) MIMO receiver DFG in PaDaF	(b) Actor Decod compute() method

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

port as shown in Figure 5(b). This format is close to SystemC [Panda 2001], using C++ to support a specific MoC.

The originality of *PaDaF* is that it permits one to describe the DFG in a closed form (or reduced graph)—that is, as a sequence of instructions describing how to build the graph. This sequence can use C++ control structure instructions (i.e., for loop) and any C++ structure for that matter. All information needed to construct the graph has to be known at compilation time. Figure 6(a) 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 Figure 4 for *NB_ANT*=4.

Each actor has a single *compute()* method that 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 in Figure 6(b).

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

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

3.2. Compiler Front End

This section deals with the front end of our compiler, which generates the intermediate representation (IR) and builds the extended DFG of the application described. The DFG construction constitutes 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 DFG construction, we use the MIMO graph construction code of Figure 6(a). This construction implies a for loop to link actors `src` to `fft`, and `fft` to `mimo`.

Figure 7 introduces a detailed view of this DFG construction. The compilation flow is based on the LLVM compiler infrastructure [Lattner and Adve 2004]. Considering that our input format `PaDaF` is based on standard C++, it takes advantage of an LLVM C++ front end—Clang [Lattner 2011]—to generate the LLVM IR. The front end illustrated in Figure 7 is decomposed in three steps: (1) construction of the extended graph, presented in Section 3.2.1; (2) identification of graph access methods in the `compute()`

methods, introduced in Section 3.2.2; and (3) linking of the graph access methods to their corresponding edge in the extended graph, 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 instantiates all actors and connects them together. The result is a set of C++ objects instantiated in memory. The compiler uses this graph representation in memory for the remainder 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 instantiated actor and a link to the `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 ports that are connected to edges of the extended graph. More precisely, they are performed by methods (e.g., `push()`) of each data port's object, as presented in Section 3.1. The first step is to isolate these method calls in the LLVM IR.

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) that appear 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 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, in Figure 6(b), access to each input port by the `Decod` actor is done through the `Iin[i]` expression; `i` might itself be the result of another expression. Rather than analyzing 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, and only a 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 Figure 6(b) 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 (SSA) form, which eases the analysis of the control flow defining the dependencies between instructions. The algorithm marks all 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 graph construction is the assumption of a static graph architecture. This constraint matches the dataflow MoC, 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 targeted platform. Next 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—that is, 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] do 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 the microscheduling technique in particular.

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 who could not foresee a 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 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, with 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 MoC. Evaluation of our compiler in terms of development time and generated code performance for the Magali platform is presented in Section 6.

4. PARAMETRIC DATAFLOW SCHEDULING

Scheduling is a key optimization problem for the efficient mapping of dataflow applications on 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 an 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 edges* 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, the 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—that is, fixed and known at compile time. A famous example of such static dataflow representation is called *synchronous dataflow* (SDF) [Bhattacharyya et al. 1999]. 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 following form:

$$\#A_1 \cdot r_{e,1} = \#A_2 \cdot r_{e,2}, \quad (1)$$

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 nonnull solutions. The minimal solution of the balance equations is called the *repetition* vector (or *iteration* vector) [Bhattacharyya et al. 1999].

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 multicore architectures [Bebelis et al. 2013a] as well as heterogeneous socs [Dardailon 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 $p, q, \dots$ that 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 Fradet et al. [2012] for a more formal definition of SPDF.

The left-hand side of Figure 8 shows an example of an 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 $2p$ executions of B . In this example, the iteration vector of

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

the graph is (A, B^{2p}, C^2, D^2) , and it usually is 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 an arbitrary parameter change period greatly complicates analysis of SPDF graphs, and of course not all parameter change period are valid. In this article, we choose (as was done in other works following SPDF [Bebelis et al. 2013a, 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 *quasistatic schedule* to refer to a schedule in which there are indications about the production and consumption of the parameter (in addition to the production and consumption of data). Although there have been other semantics associated with the term *quasistatic schedule*, we use the one of Fradet et al.: a quasistatic 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. Therefore, a quasistatic iteration vector is a repetition of quasistatic schedules of each actor possibly interleaved with the production and consumption of parameters. Because of our assumption concerning the parameter change period, it is safe to impose that each parameter consumption is performed before the execution of the actor and that each parameter production is performed after the execution of the actor. For instance, for the graph in Figure 8, the quasistatic schedule of the graph corresponding to the extension of the iteration vector with parameter synchronization is schedule (2):

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

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 a cyclic graph under certain conditions [Bhattacharyya et al. 1999]. Finding a sequential quasistatic schedule for an SPDF graph has been studied in Fradet et al. [2012], and hence in this article we assume that a valid sequential schedule exists for our applications. For instance,

schedule (3) is a valid sequential schedule obtained by a topological sort of the SPDF graph of Figure 8:

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

Such a global sequential schedule of our SPDF graph can easily be used as a starting point for finding a distributed scheduling onto a multicore platform given a specific mapping such as the one represented on the right-hand side of Figure 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 easily be built by simply scheduling each mapped actor in the order it was scheduled in the global sequential schedule.

For instance, consider the simple SPDF graph of Figure 8 executed on two IPs: IP₁ and IP₂. If A and C are mapped on IP₁ and B and D on IP₂, we obtain on schedule (4) a valid multicore schedule by scheduling on each core the actors in the order that it was scheduled in the sequential schedule:

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

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

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

4.3. Limitation of Traditional Dataflow Formalisms for Code Generation

In many works dealing with the classical SDF schedule [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 an SoC. As an example, in embedded hardware platforms, memory reserved for data communications between actors is usually very restricted. For instance, the Magali platform only allows 16 bytes of data in its fixed-size communication FIFO.

However, classical approaches with dataflow formalism make the assumption of an atomic execution of actors that is too restrictive when data transfers between actors on a real platform are concerned. Consider the example of Figure 8 with the quasistatic schedule (5) from the 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 one 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.

A few works have addressed this problem of deriving tighter lifetimes for data and thus smaller buffer sizes. Wiggers [2009] formalized a small-grain refinement of actors for parametric DFGs. 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 to radio applications with similar limitations.

In practice, actor firing does 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. The size of FIFOs can therefore be optimized further if this behavior is taken into account.

In the particular case of SPDF, parameter synchronization between quasistatic 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 of 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 microschedules as a way to explicitly express the relative dependencies between the production and consumption of data and parameters.

5. MICROSCHEDULES

In this section, we introduce our refinement to Fradet et al.'s quasistatic scheduling formalism: the microschedule formalism for parametric data flow. Then we show how to use microschedule to check the consistency between the FIFO sizes of the actual target architecture and the schedule of the actors in a more precise way.

5.1. Refining Quasistatic Schedules

The quasistatic schedule formalism was obtained by adding the production and consumption of parameters in the scheduling. We propose a second refinement that consists of adding the production and consumption of each token. This is what we call a *microschedule*. It is important to note that microscheduling is not a new MoC, but a refinement of the SPDF semantics to enable more efficient schedules on real multicore targets.

Microschedules 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 microschedule is quasistatic and known at compilation time, as was the case for all SDR IPs that we have used. The microschedule is extracted from actors' computation code for processors or predefined for hardware accelerators' IPs. One can also see the microschedule as a granularity refinement and a generalization of SDF to CSDF, as was done in Geilen et al. [2011].

Formally, the microschedule for an SPDF graph includes the following instructions in addition to the components of quasistatic schedules introduced in Section 4.2:

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

Microschedules 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 the actor schedule; however, extra care is needed for parameter I/Os that might occur in the actor schedule or IP schedules. As we saw 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. Next we explain where parameter production and consumption should be indicated.

Parameter production. A parameter is produced by an actor and should therefore be included in its microschedule and not in the schedule of the IP on which it is mapped. The simplest case is an actor A that 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 microschedule 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. Parameter consumption is provided in the IP schedule rather than in the actor microschedule 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 microschedule of the actor; however, given the fact that there is one refresh per iteration, it is safe to keep the parameter's consumption (i.e., get) outside the actor's microschedule (i.e., in the IP schedule).

A valid microschedule for the actors in Figure 8 is represented later on the left-hand side of schedule (6). Again, remember that finding a microschedule for each actor is outside the scope of this work, and most of the time it will be given in the actor or IP specification. Finding, for each actor, the best actor microschedule (e.g., to minimize the FIFO buffer size globally) is a 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 multicore scheduling. With the previous mapping, the previous multicore schedule (5) is changed to reflect the setting of parameters inside the IP schedules on the right-hand side of schedule (6).

$$\begin{aligned}
 \mu S(A) &= (set(p); (push_{AB}(1); push_{AC}(1))^{2p}) \\
 \mu S(B) &= (pop_{AB}(1); it = 1?set(q); push_{BC}(q)) \\
 \mu S(C) &= (pop_{AC}(p); push_{CD}(p)) \\
 \mu S(D) &= (pop_{BD}(q); pop_{CD}(1))^p
 \end{aligned}
 \quad
 \begin{aligned}
 S_{IP_1} &= (\mu S(A); \mu S(C)^2) \\
 S_{IP_2} &= (get(p); \mu S(B)^{2p}; \mu S(D)^2)
 \end{aligned}
 \tag{6}$$

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, the microschedule does not change the scheduling; it allows the programmer to check more precisely if a given schedule and associated microschedule 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 efficiently solve, using this microschedule formalism, the following problem: given a multicore quasistatic microschedule of an 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 microschedule to describe actor behavior in a DFG. These microschedules 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 microschedule will execute correctly with the available buffer sizes. We want to check that for *any* 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.

Spin [Holzmann 2004] is an open-source model checker targeting verification of multithreaded 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

Fig. 9. An example of a static DFG mapped on an architecture with M IPs, with 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.

(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 a mapped DFG from Figure 9 ($M = 2$, $N = 4$, $S = 3$) and the associated Promela code.

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 Figure 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 microschedule:

- Source actor A_1 : $\mu S(A_1) = (\text{push}_{A_1, A_2}(1)^N)$;
- Actor A_x , $1 < x < M$: $\mu S(A_x) = ((\text{pop}_{A_{x-1}, A_x}(1); \text{push}_{A_x, A_{x+1}}(1))^N)$;
- Sink actor A_M : $\mu S(A_M) = (\text{pop}_{A_{M-1}, A_M}(1)^N)$.

To illustrate the Promela code (Figure 10(b)), we use a particular instance of this application, depicted in Figure 10(a) and characterized by two actors ($M = 2$), four tokens exchanged ($N = 4$), and a FIFO of size 3 ($S = 3$). We want to verify that the preceding 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], which we will call the *global memory model*, or we can use the modeling that we propose hereafter, which we call the *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, whereas 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 in Figure 10(a) before formally defining the channel memory model.

Global memory model. In the first step of the execution, actor A_1 produces one 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 on dataflow scheduling [Geilen et al. 2005; Hartel and Ruys 2008; Liu et al. 2009; Malik and Gregg 2013; Dardaillon et al. 2014a] 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 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 in Figure 10(a) in the Promela code in Figure 10(b):

- Each core is encoded as a Promela process proctype. To cope with the microschedule paradigm, we are refining the dataflow MoC by removing the atomic actor execution hypothesis. All processes are marked as active, which means that they all are 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 (respectively, reading) of a single token belonging to arc y and mapped to FIFO `ch_x` is modeled by `ch_x!y` (respectively, `ch_x?y`). The `xs` (respectively, `xr`) primitive signals that the process is the only producing (respectively, consuming) tokens on the FIFO. This `chan` primitive is essential for partial order reduction 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 (respectively, consumption) of a parameter p with value y is modeled by `p_x!y` (respectively, `p_x?y`). Note that SPIN is not a symbolic model checker, and 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 Figure 10(b), SPIN attempts to verify that all execution traces lead to a correct end state—that is, all processes have ended their execution and all FIFOs are empty. If initial tokens need to be set on some edges, additional 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 performance of the two models for buffer verification, we use the application of Figure 9. Loop unrolling was applied to all global memory models to reduce the number of states of each actor. Results are presented in Figure 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 ($M \in [2 : 6]$) 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 in 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), whereas the growth is exponential for the global memory model. As mentioned previously, this can be explained

Fig. 11. Number of states stored by SPIN for the generic application from Figure 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.

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 proportional 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 the LTE protocol in Section 6.

In summary, we have shown that the association of the microschedule with model checking offers a tool to precisely control the pipeline between two IPs. In each architecture using hardware FIFOs, the sizes of these FIFOs are small (because inter-IP hardware FIFOs are 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.

Code development performance. The benefits of using our compiler are described in Table I. The estimation of the time for writing native code to Magali is not based on our experience but on the experience of engineers who programmed LTE-Advanced on Magali. As mentioned previously, 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, as its process includes reflection times that are difficult to gauge, and because it is highly dependent on the developer. However, when applications are written by people of similar technical

Table I. Comparison of Compiled Code and Handwritten Code Targeting Magali

Application	PaDaF		Native	
	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

Application	Global Memory Model		Channel Memory Model	
	Stored States	Runtime (s)	Stored States	Runtime (s)
OFDM	12,798	<0.1	5,771	<0.1
Demodulation	1.99e7	54	4,986	<0.1
Full Demod.	5.54e7	81	5,360	<0.1
Parametric Demod.	6.06e7	121	11,574	<0.1

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 Table II. These simulations were run on a 2.8GHz Intel Core i5 with 8GB 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 previously; 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.

SDF³ [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 particularly is more expressive than SPDF and has already been used to model the LTE-Advanced application [Siyoun et al. 2011]. Differences between SDF³, which is more focused on throughput, and our model, which focuses 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 one-to-one mapping between actors and cores, and buffer size of two tokens, the same size as on the Magali platform. LTE-Advanced applications were also ported to SDF³ using the platform mapping, buffer

Table III. Comparison Between SPIN and SDF on Dynamic Applications

Application	SPIN Channel Memory		SDF ³ [Theelen et al. 2011]	
	Stored States	Runtime (s)	Stored States	Runtime (s)
MPEG-4 AVC	184	<0.1	22	<0.1
Channel Equalizer	340	<0.1	297	<0.1
MPEG-4 SP (PD = 1)	5,185	<0.1	38,441	2
MP3 (PD = 1)	710,404	0.66	—	—
OFDM	5,771	<0.1	15,756	1
Demodulation	4,986	<0.1	12,002	5
Full Demod.	5,360	<0.1	6,177	1
Parametric Demod.	11,574	<0.1	52,581	215

Table IV. Performance Result of Generated Code with Respect to Handwritten Code

Application	Handwritten (μ s)	[Risset et al. 2011] (μ s)	Generated (μ s)	Optimized (μ s)
OFDM	149	500 (+236%)	168 (+13%)	149 (+0%)
Demodulation	180	—	283 (+57%)	180 (+0%)
Parametric Demod.	419	—	558 (+33%)	288 (-31%)

sizes, and microscheduling used in the channel memory model. SDF³ represents cyclic production and consumption of data as a series of states in the SADF model, which does not fare well with the idea of microscheduling. As such, each actor with a microschedule 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 SDF³ were run on the same test machine to have comparable runtime. 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 (SDF³ was run to analyze the number of states after solving nondeterminism [Theelen et al. 2011]) using SDF³. From these results, we observe a larger number of stored states and runtime due to the different type of analysis run by SDF³ 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 explained previously [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 automatizable.

The overhead of the generated approach, compared to manual code, varies from 13% for small applications up to 57% and is due to the central controller latency. To understand this latency, one has to look closer at the Magali control mechanisms [Clermidy et al. 2009a]. Each of the heterogeneous cores running the application embeds a dedicated controller, whose 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 into 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 reconfigures 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 dependent 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 performance is similar to the handwritten code for nonparametric applications and is even improved for parametric applications.

7. RELATED WORK

Various compilation flows are used to program SDR platforms, with many of them programmed using more than one language (i.e., C and assembly code, or Matlab and VHDL). On the other hand, many integrated design environments (IDES) are emerging, targeting general-purpose applications on parallel architectures or dedicated to SDR. 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 Dardailon 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, whereas 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 the 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 a library of optimized 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] and Dardailon et al. [2013] for recent surveys). The need for verifiable but still flexible dataflow MoCs has led 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 a Fradet et al. subclass of parametric dataflow—SPDF—in which 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 the quasistatic schedule needed for efficient code generation on Magali.

Concerning input language issues, one way to include complex graph construction is to rely on a template, or macro, to describe the graph and use the preprocessor to generate it at compile time, such as in the 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 the C++ programming language and executes it at compilation time. The only other dataflow language providing such complex graph construction of which we are aware is ΣC [Goubier et al. 2011], which uses a CSDF MoC without parameters and targets the MPPA homogeneous manycore platform. The complex DFG construction is handled by a new language and compilation flow, whereas 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 DFG construction. ΣC and LIME [Kourzanov et al. 2010] use arrays 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 that would not fit Magali constraints but could be considered for other more flexible platforms.

Scheduling for buffer minimization is NP-complete [Moreira et al. 2010]. Many heuristics have been developed to schedule under memory constraints [Karczmarek et al. 2003; Geilen et al. 2005; Bhattacharyya et al. 1999]. We focused on model checking solutions based on the work of 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 the buffer on the scheduled synchronous DFG. SDF³ is a reference in dataflow analysis, and we compared our work to 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 checking the liveness of a DFG 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 works 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 a 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 the microschedule enables 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 microschedules, although 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 article presents a new compilation flow, based on the LLVM framework, that compiles parametric DFGs down to heterogeneous MPSoCs. This framework is dedicated to new wireless applications using new MoCs, 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 microschedule formalism to describe actors communication behavior in DFGs. Based on this formalism, we provide a new buffer size verification method using model checking, which can be performed when mapping DFG 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 in the verification time thanks to the new verification method. The performance of the programs generated by our compiler is 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

- Vagelis Bebelis, Pascal Fradet, Alain Girault, and Bruno Lavigueur. 2013a. A framework to schedule parametric dataflow applications on many-core platforms. In *Proceedings of the 17th Workshop on Compilers for Parallel Computing (CPC'13)*.
- Vagelis Bebelis, Pascal Fradet, Alain Girault, and Bruno Lavigueur. 2013b. BPDF: A statically analyzable dataflow model with integer and Boolean parameters. In *Proceedings of the International Conference on Embedded Software (EMSOFT'13)*. 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 Annual Design Automation Conference (DAC'13)*. 3.
- Heikki Berg, Claudio Brunelli, and Ulf Lucking. 2008. Analyzing models of computation for software defined radio applications. In *Proceedings of the International Symposium on System-on-Chip*. 1–4.
- B. Bhattacharya and S. S. Bhattacharyya. 2001. Parameterized dataflow modeling for DSP systems. *IEEE Transactions on Signal Processing* 49, 10, 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, 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* 4, 155–182.
- João M. P. Cardoso, Pedro C. Diniz, and Markus Weinhardt. 2010. Compiling for reconfigurable computing. *ACM Computing Surveys* 42, 4, 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, 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 *Proceedings of the IEEE International Solid-State Circuits Conference (ISSCC'10)*. 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 *Proceedings of the 12th Euromicro Conference on Digital System Design, Architectures, Methods, and Tools*. 449–456.
- Fabien Clermidy, Romain Lemaire, and Yvain Thonnart. 2009a. A communication and configuration controller for NoC based reconfigurable data flow architecture. In *Proceedings of the 3rd ACM/IEEE International Symposium on Networks-on-Chip (NOCS'09)*. 153–162.
- M. Damavandpeyma, S. Stuijk, T. Basten, M. Geilen, and H. Corporaal. 2012. Modeling static-order schedules in synchronous dataflow graphs. In *Proceedings of the 2012 Design, Automation, and Test in Europe Conference and Exhibition (DATE'12)*. 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. 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 *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES'14)*.
- 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 *Proceedings of the Workshop on High Performance Energy Efficient Embedded Systems (HIPEAC'14)*.
- Pascal Fradet, Alain Girault, and Peter Poplavko. 2012. SPDF: A schedulable parametric data-flow MoC. In *Proceedings of the Design, Automation, and Test in Europe Conference and Exhibition (DATE'12)*. 769–774.
- Marc Geilen, Twan Basten, and Sander Stuijk. 2005. Minimising buffer requirements of synchronous dataflow graphs with model checking. In *Proceedings of the Conference on Design Automation (DAC'05)*. 819.

- Marc Geilen, Stavros Tripakis, and Maarten Wiggers. 2011. The earlier the better: A theory of timed actor interfaces. In *Proceedings of the International Conference on Hybrid Systems: Computation and Control (HSCC'11)*. 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 *Proceedings of the Conference on Formal Methods in Computer Aided Design (FMCAD'06)*. IEEE, Los Alamitos, CA, 68–75.
- C. R. A. Gonzalez, C. B. Dietrich, S. Sayed, H. I. Volos, J. D. Gaedtert, 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, 48–55.
- Jair Gonzalez-Pina, Rabea Ameur-Boulifa, and Renaud Pacalet. 2012. DiplodocusDF, a domain-specific modelling language for software defined radio applications. In *Proceedings of the 38th Euromicro Conference on Software Engineering and Advanced Applications*. 1–8.
- Thierry Goubier, Renaud Sirdey, Stéphane Louise, and Vincent David. 2011. ΣC : A programming model and language for embedded manycores. In *Proceedings of the 11th International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP'11)*. 385–394.
- Pieter H. Hartel and Theo C. Ruys. 2008. Scheduling optimisations for SPIN to minimise buffer requirements in synchronous data flow. In *Proceedings of the International Conference on Formal Methods in Computer-Aided Design (FMCAD'08)*.
- Julien Heulot, Maxime Pelcat, Jean-François Nezan, Yaset Oliva, Slaheddine Aridhi, and Shuvra S. Bhat-tacharyya. 2014. Just-in-time scheduling techniques for multicore signal processing systems. In *Proceedings of the Global Conference on Signal and Information Processing (GlobalSIP'14)*. 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 *Proceedings of the International Conference on Formal Description Techniques (FORTE'94)*. 197–211.
- W. M. Johnston, J. R. Paul Hanna, and R. J. Millar. 2004. Advances in dataflow programming languages. *ACM Computing Surveys* 36, 1, 1–34.
- S.-H. Kang, H. Yang, L. Schor, I. Bacivarov, S. Ha, and L. Thiele. 2012. Multi-objective mapping optimization via problem decomposition for many-core systems. In *Proceedings of the 2012 IEEE 10th Symposium on Embedded Systems for Real-Time Multimedia (ESTIMedia'12)*. 28–37.
- Michal Karczmarek, William Thies, and Saman Amarasinghe. 2003. Phased scheduling of stream programs. In *Proceedings of the Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'03)*. 103.
- Pjotr Kourzanov, Orlando Moreira, and Henk J. Sips. 2010. Disciplined multi-core programming in C. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'10)*. 346–354.
- Seongnam Kwon, Yongjoo Kim, Woo C. Jeun, Soonhoi Ha, and Yunheung Paek. 2008. A retargetable parallel-programming framework for MPSoC. *ACM Transactions on Design Automation of Electronic Systems* 13, 3, 1–18.
- Chris Lattner. 2011. LLVM and Clang: Advancing compiler technology. In *Proceedings of the Free and Open Source Developers' European Meeting (FOSDEM'11)*. 234–239.
- Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis and transformation. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'04)*. 75–86.
- 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 *Proceedings of the SDR Forum Technical Conference*. 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 *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS'09)*. 61–70.
- Avinash Malik and David Gregg. 2013. Orchestrating stream graphs using model checking. *ACM Transactions on Architecture and Code Optimization* 10, 3, 19.
- Kevin Marquet and Matthieu Moy. 2010. PinaVM: A SystemC front-end based on an executable intermediate representation. In *Proceedings of the International Conference on Embedded Software (EMSOFT'10)*.
- Orlando Moreira, Twan Basten, Marc Geilen, and Sander Stuijk. 2010. Buffer sizing for rate-optimal single-rate data-flow scheduling revisited. *IEEE Transactions on Computers* 59, 2, 188–201.
- Preeti Ranjan Panda. 2001. SystemC—a modeling platform supporting multiple design. In *Proceedings of the 14th International Symposium on Systems Synthesis (ISSS'01)*. 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 *Proceedings of the European Embedded Design in Education and Research Conference (EDERC'14)*. 36–40.
- P. Quinton and T. Risset. 2001. Structured scheduling of recurrence equations: Theory and practice. In *Embedded Processor Design Challenges*. Lecture Notes in Computer Science, Vol. 2268. Springer, 112–134.
- Tanguy Risset, Riadh Ben Abdallah, Antoine Fraboulet, and Jérôme Martin. 2011. Programming models and implementation platforms for software defined radio configuration. In *Digital Front-End in Wireless Communications and Broadcasting*. Cambridge University Press, 650–670.
- Hrishikesh Salunkhe, Orlando Moreira, and Kees van Berkel. 2014. Mode-controlled dataflow based modeling and analysis of a 4G-LTE receiver. In *Proceedings of the Conference on Design, Automation, and Test in Europe (DATE'14)*. 212.
- Firew Siyoum, Marc Geilen, Orlando Moreira, Rick Nas, and Henk Corporaal. 2011. Analyzing synchronous dataflow scenarios for dynamic software-defined radio applications. In *Proceedings of the International Symposium on System on Chip (SoC)*. IEEE, Los Alamitos, CA, 14–21.
- Sander Stuijk, Marc Geilen, and Twan Basten. 2006. SDF3: SDF for free. In *Proceedings of the International Conference on Application of Concurrency to System Design (ACSD'06)*. IEEE, Los Alamitos, CA, 276–278.
- Sander Stuijk, Marc Geilen, Bart Theelen, and Twan Basten. 2011. Scenario-aware dataflow: Modeling, analysis and implementation of dynamic applications. In *Proceedings of the International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation*. 404–411.
- Bart Theelen, Marc Geilen, and Jeroen Voeten. 2011. Performance model checking scenario-aware dataflow. In *Formal Modeling and Analysis of Timed Systems*. Lecture Notes in Computer Science, 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 *Proceedings of the 2012 Real-Time and Embedded Technology and Applications Symposium (RTAS'12)*. 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 *Proceedings of the 33rd International Symposium on Computer Architecture (ISCA'06)*. 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 *Proceedings of the Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation*. 343–354.
- J. Zyren and W. McCoy. 2007. *Overview of the 3GPP Long Term Evolution Physical Layer*. Technical Report. Freescale Semiconductor Inc.

Received August 2015; revised March 2016; accepted March 2016