Compilation for heterogeneous SoCs : bridging the gap between software and target-specific mechanisms

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

Compilation for heterogeneous SoCs : bridging the gap between software and target-specific mechanisms.

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

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

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

ABSTRACT

Current applications constraints are pushing for higher computation power while reducing energy consumption, driving the development of increasingly specialized SoCs. In the mean time, these SoCs are still programmed in assembly language to make use of their specific hardware mechanisms. The constraints on hardware development bringing specialization, hence heterogeneity, it is essential to support these new mechanisms using high-level programming. In this work, we use a parametric data flow formalism to abstract the application from any hardware platform. From this premise, we propose to contribute to the compilation of target independent programs on heterogeneous platforms. These developments are threefold, with 1) the support of hardware accelerators for computation using actor fusion, 2) the automatic generation of communications on complex memory layouts and 3) the synchronization of distributed cores using hardware mechanisms for scheduling. The code generation is illustrated on a telecommunication dedicated heterogeneous SoC.

Categories and Subject Descriptors

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

1. INTRODUCTION

Throughout its history, processor development has been driven by the Moore law, but was also by technology constraints. The *power wall*, whereby instruction level parallelism was not improved effectively by increasing transistor count and frequency growth as halted, was answered by the apparition of multi-core processors. We now enter the *dark silicon era* [8], which fixes a limit on the number of transistors that can be powered on at a given time. To answer this new limitation, chip designers are specializing parts of their design for application-specific constraints (e.g. video decoding, voice recognition), which leads to more and more heterogeneous System on Chip (SoC) [3]. We now find hardware acceleration not only in computation, but also in communication and control, making it ubiquitous and which must be programmed efficiently.

From the software point of view, programming efficiently each new kind of architecture is a renewed challenge. The multiplication of cores requires explicit parallelism in the application, with several solutions from imperative concurrent models (e.g. Pthreads, MPI), but also approaches like the data flow model of computation (MoC). The emergence of heterogeneity brings a new constraint, each platform requiring specific instructions in order to take advantage of the specialized parts.

Our problem is to find how to program these new heterogeneous platforms efficiently. This efficiency has to be found both in terms of raw computing performance, but also in terms of time and difficulty to write new programs or adapt existing programs to new platforms. To reach this objective, a high-level language is necessary, current targets being too complex to be programmed using assembly language. Another constraint is the platform abstraction for both having a portable implementation and reducing the requirements on the platform mechanisms knowledge.

Another problem is that the targeted applications on these platforms adds both performance and timing constraints. As an example, telecommunication protocol LTE-Advanced requires 40 $GOPS$ and a latency less than 2 μs [5]. In order to achieve the performance target, software optimizations can be done both at compilation time and at runtime. Runtime optimizations are able to reach high throughput by leveraging runtime knowledge, at the cost of an initial overhead. This overhead is usually made profitable given compatible time constraints, but is clearly not possible within our latency requirements.

To answer the high-level representation and parallelism constraints, we propose to use a parametric data flow MoC. In this model, data and task parallelism are exposed to the compiler. Moreover, this MoC permits many static analyses and associated optimizations to match the constraints of our target application.

In this paper, we propose compilation methods which do not exist in current state of the art data flow compilers. Our contribution is threefold :

- Platform independent language primitives thanks to the use of actor fusion.
- Compilation of parametric inter-core communications.
- Code generation for distributed scheduling targeting specialized controllers.

The remaining of the paper is built as follow: we describe our target demonstrator as well as related work on heterogeneous SoC compilation in section 2 to motivate our work; the compilation MoC and language are introduced in section 3; contributions are described in section 4; results sustaining our approach are presented in section 5 before the conclusion.

2. MOTIVATION AND RELATED WORK

To better understand the characteristics of heterogeneous SoCs targeted by our compilation flow, we look at the CEA Magali platform [6], before reviewing related work in heterogeneous programming.

2.1 Heterogeneous SoC example : Magali

The Magali chip [6] is a system on chip dedicated to physical layer processing of OFDMA radio protocols, with a special focus to 3GPP LTE-Advanced as reference application. It includes heterogeneous computation hardware, with very different degrees of programmability, from configurable blocks to DSPs programmable in C. As an example, one reconfigurable block is used to perform both an FFT and a deframing (removing some of the resulting data). This operator permits to preserve only the significant data, hence reducing the data transmission time, but is also platform specific and needs to be abstracted away.

Communications between blocks use a 2D-mesh network on chip (Network-on-Chip). All the data communications are programmed statically on a credit/data mechanism between source and destination called ICC for input, OCC for output. One big difficulty when programming this platform is to guarantee consistency for all communications between all blocks, which for non trivial applications makes manual writing a daunting task. The example on Fig. 1 illustrates this on a toy application with 4 cores. In this example, core `A` sends 30 data to core `C` using OCC 0. The OCC configuration for sending data needs to know which ICC it addresses and how many data it sends. Likewise, the ICC 0 on core `C` needs to know how many credits to send and to which OCC. On this simple application, coherency between 10 configurations has to be guaranteed by the programmer, making it error prone.

Main configuration and control of the chip is done by an ARM CPU. Magali offers distributed control features, enabling to program sequences of computations at core level. Distributed Configuration and Communication Controllers (CCC) support program sequences, with two levels loops and automatic program memory caching. More details can be found in [7]. The controllers in charge of the program sequences are limited in scope and platform specific, but also essential to efficiently use Magali. They are illustrated in Fig. 1. Core `C` distributed controller is programmed to repeat two times a first configuration, before switching to a second configuration. Using the same distributed control mechanism, communications sequences are also programmed for each input/output.

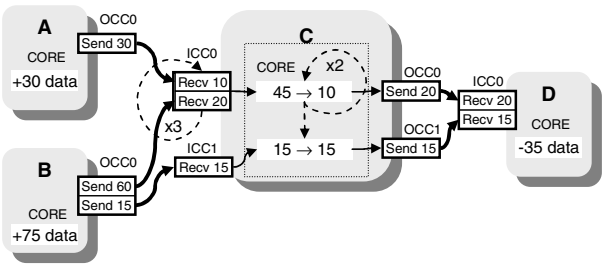


Figure 1: Data and processing flows example in Magali (from [7])

Looking at this example platform, we see that computation, communication and control are hardware accelerated and need to be supported in the programming flow. In the next section, we look at some significant works in the domain of heterogeneous SoC programming from the target perspective, starting from general solutions before focusing on data flow programming.

2.2 Related work

For an embedded software programmer, the easiest way to program an heterogeneous platform is to use an imperative language (generally `C` language) associated with threads to express parallelism. It has been used to program both heterogeneous and homogeneous parallel platforms. For instance, the different units of the BEAR SDR platform [15] are programmed using `C` and Matlab code.

The ExoCHI [19] programming environment and Merge [13] framework (based on ExoCHI) are proposals aiming at easing the programming of heterogeneous platforms while achieving good performances. The proposed solution extends OpenMP with intrinsic functions and dynamically maps the software on available resources. Similarly, OpenCL [12] can be used for heterogeneous platforms support.

The main limitation of these approaches is their lack of abstraction, requiring to program explicitly all hardware mechanisms. Even if using a high-level representation, the programmer still needs precise knowledge of the platform's specific resources, in order to write the required platform-specific program.

Numerous research works present arguments in favor of a paradigm shift and propose to program waveforms using data flow languages. These languages relies on a data flow *Model of Computation* (MoC) where a program is represented as a directed graph $G = (V, E)$. An actor $v \in V$ represents a computational module or a hierarchically nested subgraph. A directed edge $e \in E$ represents a FIFO buffer from its source actor S to its destination actor D. Data flow graphs follow a data-driven execution: an actor v can be

executed (fired) only when enough data samples are available on its input edges. When firing, v consumes a certain amount of samples from its input edges and produces a certain number of samples on its output edges.

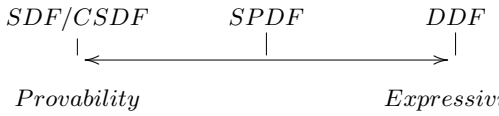


Figure 2: Representation of the balance between provability and expressivity in data flow computation models.

Many data flow-compliant programming models have been proposed for specific applications; they are illustrated in Fig. 2. *Synchronous Data Flow* (SDF) means that the number of tokens necessary for an actor to fire is known at compile-time. In this case, static scheduling of actors can be performed and the size of the buffers between actors can be bounded. In *Dynamic Data Flow*, data samples consumed and produced by an actor at each firing can vary dynamically at runtime, and can even be 0 in order to provide more flexibility for programming. As a drawback, theoretical analysis capabilities are reduced. Between synchronous and dynamic data flow formalisms, a wide amount of models have been proposed, e.g. *Cyclo-Static Data Flow* (CSDF) [2], Schedulable Parametric Data Flow (SPDF) [9]. The goal was to look for a trade-off between the ability to statically analyze programs and the expressivity of the languages.

ΣC [11] is a proposal to program waveforms using an extension of `C`. The corresponding MoC is more expressive than SDF thanks to non-deterministic extensions but still allows some static analyses to be performed such as bounding memory usage. However it does not allow dynamic behavior of actors. MAPS [4] is also based on a `C` extension, and uses a dynamic data flow MoC. The dynamicity is supported through a dynamic mapping, with high level operators mapped on accelerators at runtime. We can also mention work on the Magali platform using the KPN MoC [14]. The mapping is done on abstract architecture and platform to reduce the platform dependency.

One common pattern in these approaches is the use of `C` language or a derivative to represent computation in a familiar manner, while using high level concepts such as threads or actors to convey parallelism. Platform specific operators are supported by libraries and APIs, with some environments such as ExoCHI allowing to split operators to fit the platform granularity.

The recent progress of significative works such as ExoCHI and MAPS testifies to the importance of research in heterogeneous SoC programming. However, these approaches uses dynamic MoCs which are supported by multi-core processors, but falls outside the scope of our platform. We propose to work on a parametric data flow MoC to get the desired expressivity while keeping analyzability. Starting from this premise, we put forward a new compilation flow and runtime for this MoC.

3. FRAMEWORK

3.1 Model of computation

In this work, we chose to start from a data flow MoC to harness its inherent parallelism and analyzability. Moreover, the need for verifiable but still flexible data flow MoCs recently lead to the appearance of two new MoCs: Scenario-Aware Data Flow [18] and Parametric Data Flow [9]. Fradet and Girault identify a subclass of this MoC called *schedulable parametric data flow* (SPDF) where the schedulability of the data flow graph can still be assessed statically. We chose to experiment using this MoC, but most of the contribution remains valid for analyzable data flow models. Moreover, SDF being a subset of SPDF without parameters, our work and the resulting compiler applies to SDF.

To introduce SPDF, a program is represented on Fig. 3 with four data flow actors named `A,B,C` and `D`. As in a classical data flow graph, the integers on the arcs represent the number of samples produced or consumed by the actor at each execution. In SPDF, this number can be a symbolic parameter, the value of which is produced by an actor (e.g. the *set p[1]* in the left actor of Fig. 3).

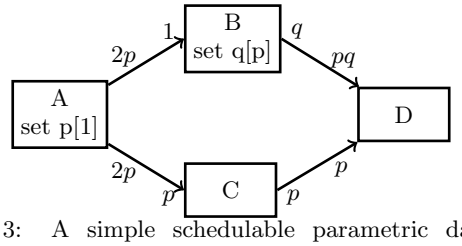


Figure 3: A simple schedulable parametric data flow graph [9], p and q are *parameters* instantiated at execution.

One interesting feature of SPDF is the ability to generate one schedule per actor, which is repeated at each iteration of the graph. The data flow graph being parametric, so is the number of iterations, and the generated schedule is a *quasi-static schedule*. As an example, the schedule for actor `B` in Fig. 3 is $(pop\ p; (B^p; push\ q)^2)$, which reads as: get parameter p, then repeat twice the following: execute p times the core code of B, provide parameter q (computed in the core code of B). The reader is invited to refer to [9] for implementation details.

3.2 Language

The language we propose is based on C++. It consists of a set of classes allowing to describe a parametric data flow graph. Fig. 4 illustrates the language on a small example.

Each actor has a single `compute` method that represents the execution of one iteration of the actor. The code of this method is written in C++ and uses various `push`/`pop` intrinsics to send/receive data, as well as `set`/`get` for parameters. An intrinsic API is also used for each application domain. It includes common, platform agnostic operations such as `fft`.

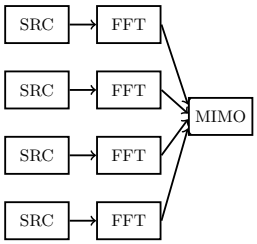
Choosing C/C++ language for the core code of the actors presents many advantages: it allows to reuse legacy code and highly optimized tools such as C compilers, do not require to learn a new language, and permits easy simulation and functional validation. Moreover, the support of a general purpose language for describing the graph *structure* greatly simplifies the development of complex applications.

```
void MIMO::compute() {
[...]
for(i=0; i<nb_ant; i++) {
  val[i]=in[tab[i]].pop();
}
[...]
```

(a) Core computation

```
SRC src[NB_ANT];
FFT fft[NB_ANT];
MIMO mimo(NB_ANT);
for(i = 0; i<NB_ANT; i++){
  fft[i].in <= src[i].out;
  mimo.in[i] <= fft[i].out;
}
```

(b) Graph code



(c) Graph constructed

Figure 4: A simple program excerpt and the corresponding part of the graph for `NB_ANT`=4.

Having a graph representing all computations and communications is half of the work. We will explain in the next section how we use this representation to generate code for hardware accelerated computation, communication and scheduling.

4. CONTRIBUTIONS

4.1 Computation

In section 2.2 we have seen that support for hardware acceleration can be done using an API, which is the case in our approach. The difficulty to solve here is when a hardware accelerator covers several operations of the API. In that case the operations may be distributed into several actors, possibly preventing to take full advantage of the accelerator's capabilities. To solve this problem, we chose to repurpose the notion of *actor fusion* to support hardware acceleration.

Filter fusion was first introduced by Proebstring et al. [16], and adapted to data flow in StreamIt [10]. Fusion is a transformation which assembles together two actors by merging their compute functions, enabling optimizations inside the resulting function. In previous works, optimization consists in inlining the code of the two actors, replacing FIFOs by local variables. This transformation enables code optimization on the resulting function, with buffer minimization or loop unrolling. In our case, it also enables the support of complex hardware accelerators by merging intrinsic calls in the same actor compute function, allowing the resulting actor to match the hardware granularity. We show below how to support SPDF actor fusion and set conditions for fusing actors in an arbitrary complex graph.

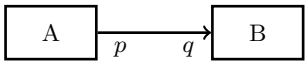


Figure 5: Simple parametric graph example.

Fig. 5 illustrates a minimal parametric graph with two actors `A` and `B` we want to fuse. When fusing actors in SDF, rates between the two actors are used to compute the execution order of the resulting actor. Contrary to SDF programs where rates are statically known, here p and q are parametric expressions unknown at compilation time.

To fuse two actors in a parametric graph, we use the concept of *local solution* from SPDF [9] to solve the execution order. The *local solution* defines for a subset of actors the minimal number of iterations of an actor to return the subset to its initial state, and is denoted $\#_L X$. Using these local solutions, code is generated for the composite actor.

```
void AB::compute() {
  data_t fifo_AB[p*q];
  for(i=0; i < q; i++) {
    A::compute()
  }
  for(i=0; i < p; i++) {
    B::compute()
  }
}
```

Figure 6: Resulting code of actor `A` and `B` fusion in Fig. 5.

The code resulting from fusion of actors `A` and `B` from Fig. 5 is illustrated in Fig. 6. The compute function of each actor X_i is inlined inside a loop repeated $\#_L X_i$ times, with $\#_L A = q$ and $\#_L B = p$ in our example. Local variables replaces FIFO connecting actors. It is worth noting that push and pop access to this FIFO in `A::compute()` and `B::compute()` are replaced by access to local variable `fifo_AB`. At this point, code optimization is applied on the generated function.

On the graph level, a composite actor is created with the two fused actors. The FIFO between actors have been replaced, and others input/output FIFOs are connected to the resulting actor.

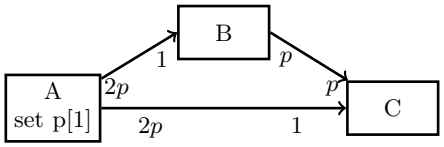


Figure 7: Fusion example of actors `A` and `C` would lead to a deadlock.

Safe actor fusion is conditioned by the graph topology, some fusions possibly creating a graph deadlock. For example, in Fig. 7 the fusion of actors `A` and `C` would create an interdependency between resulting actor `AC` and `B`. Indeed, `AC` is waiting for data on edge (B,C) to start, while actor `B` is waiting for data on edge (A,B). From this interdependency ensue a deadlock between `AC` and `B`. In order to avoid such deadlock, we define a criterion for fusion safety.

DEFINITION 1 (FUSION SAFETY). *Let $P(X_i, X_j)$ be the set of paths connecting actors X_i and X_j. The fusion between two actors X_i and X_j is safe if all paths between the two actors are a direct edge between them. Formally,*

$$\forall p \in P(X_i, X_j), \quad p = (X_i, X_j)$$

Intuitively, this condition prevents the creation of new loops in the graph, effectively avoiding interdependency creation resulting in a deadlock. A similar condition is present in StreamIt, where vertical fusion is limited to *pipeline* construct [10].

Feedback loops are not accounted in our safety criteria, which ignores edges with initial tokens. Indeed, edges used for

feedback loops have initial tokens to permit execution of the graph cycle. The liveness property defined in SPDF [9] ensures that enough tokens are available on each feedback edge to avoid deadlock. On this condition, the graph is said to be *loosely connected*, and we can safely ignore edges containing initial tokens in our fusion criterion.

Based on fusion mechanism, the mapping of actors on cores is responsible for the support of hardware mechanisms. If two actors are using API calls which are supported by a single core, these two actors are mapped on the same core and fused. Code specialization is applied in the resulting actor to match the platform operator granularity, optimizing hardware resources. Currently, due to resources constraints, the mapping in our compilation flow is provided manually by the programmer. Automation of this mapping could rely on a model of the platform cores and their affinity to API calls, which would take advantage of the platform heterogeneity. We describe concrete fusion example on our experimentation platform in section 5.

4.2 Communication

All communications in a data flow graph are explicit by construction, actors having no side effects. Using this property, we know statically all communications, with their sources, destination and the (potentially parametric) number of data exchanged. From this analysis, we can guarantee that the data flow graph can be executed in bounded time, with bounded memory, or send feedback to the programmer as to which part of his application is defective.

In our compilation flow, actors are statically allocated to cores. Using this mapping, FIFO memory can be allocated locally to minimize the memory access cost. We note that application graphs have no prerequisite on the platform memory layout. From this observation, we conclude that our data flow application is platform-agnostic. Such a property is very important, as it unlock programming of very different platforms with possibly complex memory layouts, without putting the burden on the developer.

Moreover, using the communication and placement information, we are able to generate all configurations for communication hardware mechanisms automatically. Compared with of the difficulty encountered when writing these configurations manually, as described in section 2.1, the data flow abstraction is a real game changer. The added bonus of this method is that communications are verified in the data flow graph, and the generated code is correct by construction.

4.3 Scheduling

Distributed scheduling of tasks is a hard problem to tackle by itself. The distributed controllers add to this complexity by requiring a specific model of execution (MoE), potentially different for each targeted platform. To solve this problem, we use the *quasi-static schedule* from SPDF.

This schedule model is interesting as it makes no assumptions on the underlying MoE. Looking at the bibliography, we back-up that claim with implementation on several platforms with very different MoE.

Dynamic scheduler. The authors of SPDF propose a first model for parameter communications, using up and down samplers [9]. Their model can be scheduled as a dynamic data flow graph by existing approaches.

Slotted scheduler. In another approach on the STHORM platform [1], the parametric data flow graph state is used to schedule actors in time slots. This approach could easily be adapted to use quasi-static schedules instead of the graph.

Split scheduler. We propose to use a new approach, splitting the schedule between central and distributed controllers, to demonstrate the versatility of our approach for platforms such as Magali.

To illustrate this splitting, we use the schedule of actor `B` from Fig. 3, $(pop\ p; (B^p; push\ q)^2)$. In this schedule, B^p is managed by the distributed controller, while parameter communication is managed by the central controller. For a large parameter p this distributed scheduling saves a lot of potentially costly synchronizations.

5. RESULTS

In the previous section we described how we abstract hardware acceleration using a data flow MoC, especially for computation, communication and scheduling. We are now going to demonstrate the viability of our approach on the CEA Magali platform, answering the needs from section 2.1. We extracted representative parts of the LTE protocol to illustrate the challenges in terms of programmability and dynamicity, and split them into 3 test case applications.

The mapping of the applications on the Magali architecture is illustrated in Fig. 8a. The *FFT test case* in Fig. 8b is a simple application demonstrating the support of platform specific operators. The *parametric demodulation test case* is presented in Fig. 8c. The upper part corresponds to demodulation of a LTE frame header, which enables determination of the modulation scheme `mod` used to demodulate the remaining of the frame, as shown on the bottom part of the figure. An intermediate test case has also been set-up, which corresponds to the first part of previous test case, i.e. without the actors that depends on `mod`.

5.1 Computation

The FFT test case showcases the support of platform specific accelerators using abstract operators because the OFDM core is capable of performing both an FFT and a deframing operation, which are implemented using two actors in the application graph. The fusion of these two actors enables full support of the OFDM core. The fusion is also used on the SME core in the parametric demodulation. The split actor mapped to the SME is only doing data manipulation and can be processed by the OCC mechanism on the SME core, reducing the required number of FIFOs. More complex fusions involving parameters have also been performed, for example on the RX BIT core in the parametric demodulation test case.

Besides, to tackle the lack of support for variable modulation

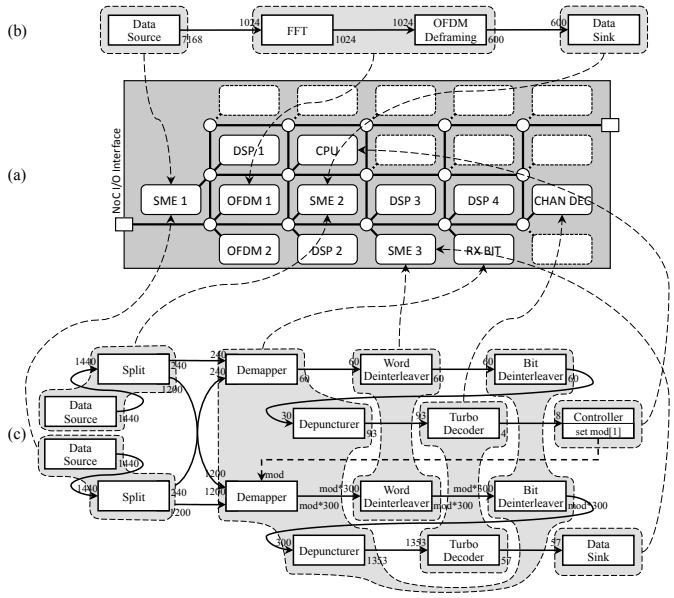


Figure 8: Mapping of the test case applications on Magali.

offered by the RX BIT core, another feature has been introduced in our compilation chain: to address the translation of the `mod` parameter into real RX BIT core configuration, and since the parameter has a limited set of possible values, the compilation flow generates for each value the corresponding configuration. Control code is automatically inserted in the scheduling to choose the correct configuration depending on the parameter value at runtime. Use of this feature is of course possible for other parameters and cores.

5.2 Communication

Reminding the constraints of the platform exposed in section 2.1, all communications on the Magali platform need to be statically generated. As we have seen, data flow exposes all communications, and we are able to assess statically for each communication the cores involved, the number of data sent, the parameters used and their producer. Using this knowledge, generation of the configurations for all ICC/OCC communications accelerators is automatic and suffers none of the complexity of the hand written approach, while resulting to similar performances (see section 5.4).

5.3 Scheduling

Once computation and communication are generated, the next step consist in scheduling them. From section 2.1, we know the Magali platform offers limited distributed scheduling capabilities. Expressivity of the quasi-static scheduling is supported by splitting the control between the central controller (an ARM processor) and the distributed controllers (CCC), as illustrated in the *split scheduler* approach from section 4.3.

To make this splitting clear, we reuse the schedule example of actor `B` in Fig. 3: $(pop\ p; (B^p; push\ q)^2)$. The CCC cannot produce or consume parameters, excluding *pop p* and *push q* commands. These steps are performed by the central controller. In turn, this controller sends the B^p schedule to the CCC for distributed scheduling when knowing the p value.

The split quasi-static schedules are generated automatically by our compiler and benefit from the Magali platform specific support for parameters, which was not the case in previous works. This shows the relevance of the SPDF to address distributed hardware constraints.

In terms of execution model, the central controller manages schedules for all cores using one thread per core. The synchronization with the distributed hardware is provided by interrupts and semaphores from the ECOS operating system.

5.4 Performances

To assess the performances obtained using our compiler, we present execution times in Tab. 1. Results are compared between hand-written code, generated code and a previous work on the Magali platform [17] for the first test case. This work uses a *radio virtual machine* to brings portability and flexibility, at the expense of byte code interpretation overhead. Moreover, the physical layer description model used is too simple to effectively take advantage of the distributed control features offered by Magali, resulting in an important increase of execution time.

Application	hand-written	generated	[17]
FFT	149 μs	168 μs (+13%)	500 μs (+236%)
demodulation	180 μs	283 μs (+57%)	-
parametric demodulation	288 μs	558 μs (+94%)	-

Table 1: Performance result of generated code with respect to hand-written code

Indeed, the results on the FFT test case showcase the need for distributed controller support, with ten times less overhead in our approach than in [17]. For the two others test cases, the platform is correctly supported with our abstraction and applications are running faultlessly. The main reason for the overhead in the compiled application is the MoE used on the central controller. Typically, hand written code is synchronized on parameter exchange at the graph level, splitting the application in so-called *phases*. Our compiler generates a dedicated thread for each core on the central controller, aiming at a finer-grained control. This fine-grained control induces a higher number of synchronizations on the central controller, resulting in an important overhead.

It should be noted that the MoE used in Magali is independent from the compilation flow. In order to improve performances on this target, ongoing work is done to improve this model to fit the platform constraints. This can be done on the one hand by specializing the scheduler for the types of synchronizations required by data flow, on the other hand a better use of static knowledge harvested on the graph to reduce the number of synchronizations. This can be done while maintaining the same compilation flow, and will further prove the relevance of our approach.

6. CONCLUSION

Current architectural trend, especially in embedded applications, is leaning towards heterogeneous, specialized SoCs to answer energy and performance constraints. Programming these platforms is inherently difficult, with a lack of adapted

tools and abstractions. New data flow MoCs such as SPDF provide sufficient expressivity to describe the complex applications targeted by these platforms, as well as a relevant abstraction level to ease the programmer's work.

Based on that premise, we have shown how the analysis made on SPDF can be used to generate code that takes benefits from the acceleration hardware mechanisms found in nowadays SoCs. We described the framework provided by our compilation flow for analysis on computation, communication and scheduling. Using these informations, we demonstrated the viability of our approach on the Magali LTE modem prototype for each of these hardware mechanisms, backed by performances results on a real-world application. Adapting this compilation flow to other heterogeneous platforms is currently ongoing, as well as optimization of the central scheduler to reach better performance on Magali.

Acknowledgment

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

7. REFERENCES

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

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

[3] D. Burger. The Golden Age of Computer Architecture May Be Now. In *High Performance Embedded Architectures and Compilers, 8th International Conference, HiPEAC*, Berlin, Germany, Jan. 2013.

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

[5] F. Clermidy, C. Bernard, R. Lemaire, J. Martin, I. Miro-Parades, Y. Thonnart, P. Vivet, and N. Wehn. A 477mW NoC-Based Digital Baseband for MIMO 4G SDR. In *Solid-State Circuits Conference (ISSCC), 2010 IEEE International*, pages 2008–2010, San Francisco, CA, Feb. 2010.

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

[7] F. Clermidy, R. Lemaire, and Y. Thonnart. A Communication and configuration controller for NoC based reconfigurable data flow architecture. *3rd ACM/IEEE International Symposium on Networks-on-Chip*, pages 153–162, 2009.

[8] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. In *Proceeding of the 38th annual international symposium on Computer architecture*, page 365, San Jose, CA, June 2011.

[9] P. Fradet, A. Girault, and P. Poplavko. SPDF: A Schedulable Parametric Data-Flow MoC. In *Design, Automation and Test in Europe international conference*, pages 769 – 774, Dresden, Germany, Mar. 2012.

[10] M. I. Gordon, D. Maze, S. Amarasinghe, W. Thies, M. Karczmarek, J. Lin, A. S. Meli, A. a. Lamb, C. Leger, J. Wong, and H. Hoffmann. A stream compiler for communication-exposed architectures. *ACM SIGARCH Computer Architecture News*, 30(5):291, Dec. 2002.

[11] T. Goubier, R. Sirdey, S. Louise, and V. David. ΣC: A programming model and language for embedded manycores. *Algorithms and Architectures for parallel processing*, pages 385–394, 2011.

[12] P. O. Jääskeläinen, C. S. d. L. Lama, P. Huerta, and J. H. Takala. OpenCL-based design methodology for application-specific processors. In *Embedded Computer Systems, 2010 International Conference on*, pages 223–230, Samos, Greece, July 2010.

[13] M. D. Linderman, J. D. Collins, H. Wang, and T. H. Y. Meng. Merge : A programming model for heterogeneous multi-core systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 287–296, Mar 2008.

[14] A. E. Mrabti, H. Sheibanyrad, F. Rousseau, F. Pétrot, R. Lemaire, and J. Martin. Abstract Description of System Application and Hardware Architecture for Hardware/Software Code Generation. In *Digital System Design, Architectures, Methods and Tools. 12th Euromicro Conference on*, pages 567 – 574, Patras, Greece, 2009.

[15] M. Palkovic, P. Raghavan, M. Li, A. Dejonghe, L. Van der Perre, and F. Catthoor. Future Software-Defined Radio Platforms and Mapping Flows. *Signal Processing Magazine, IEEE*, 27(2):22–33, 2010.

[16] T. A. Proebsting and S. A. Watterson. Filter fusion. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '96, pages 119–130, 1996.

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

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

[19] P. H. Wang, J. D. Collins, G. N. Chinya, H. Jiang, X. Tian, M. Girkar, N. Y. Yang, G.-Y. Lueh, and H. Wang. Exochi: architecture and programming environment for a heterogeneous multi-core multithreaded system. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 156–166, 2007.