Radio Science For Humanity

Software stack for code generation of software-defined radio
Pile logicielle pour la génération de code au service de la radio logicielle

***Adrien Canuel*[1], *Renaud Pacalet*[1], *and Ludovic Apvrille*[1]**

[1] *Institut Mines-Telecom, Telecom ParisTech, nom.prénom@telecom-paristech.fr*

Keywords: IoT, signal processing, code generation
Mots-clefs: IoT, traitement du signal, génération de code

Abstract:
Hardware architecture for signal processing have an increasing heterogeneity and parallelism in order to cope with the real time performance and low consumption problematics. This complexity has an influence on the development of application running on those platforms. To ease the development processes, modeling tools already exists. Our laboratory created its own modeling tool, which can generate a simulation code from the mode l designed. The work presented in this article describe an abstraction methodology of the hardware for this tool, so that the code generated by this tool can run directly on the target platform. This methodology relies on the definition of data structures and functions directly included in the code by the generator, as well as services that must be provided by the runtime environment.

Résumé:
Les architectures matérielles pour le traitement du signal sont de plus en plus hétérogènes et parallèles afin de pouvoir concilier les problématiques de performance temps réel et de consommation d'énergie. Cette complexité se répercute sur le développement des applications destinées à s'exécuter sur ces plate-formes. Afin de pouvoir faciliter le processus de développement, des outils de modélisation ont déjà été développés. Notre laboratoire a créé son propre outil de modélisation, et propose déjà de pouvoir générer du code de simulation à partir des modèles réalisés. Les travaux présentés dans cet article montrent une méthodologie d'abstraction du matériel destinée à se greffer à cet outil, de fa con à ce que le code généré puisse être directement exécutable sur l'architecture cible. Cette méthodologie repose sur la définition d'un certain nombre de structures de données et de fonctions directement intégrées dans le code par le générateur, et de services que l'environnement d'exécution doit fournir.

1 Introduction

Last generation integrated circuits for digital signal processing are widely used in communication devices, from the base stations of mobile network to the user terminals and all the intermediate elements (*small cells*, relay nodes, etc.) They progressively replace fully wired hardware solutions, composed of independent modules dedicated to an unique radio interface. They offer real opportunities in terms of energy efficiency, cost or flexibility thanks to a delicate balance between wired, reconfigurable and programmable functions.

Those circuits include general purpose microprocessors, micro controllers, specialized microprocessors for real time control, digital signal processing units (DSP) and hardware accelerators dedicated to a specific processing or to a family of processing (channel decoding, vector computing, etc.) All these units communicate with various channels, from the simplest control buses to high performance networks on chip. Finally, memory is often distributed between low latency but small local memories and larger memories with longer latency (non-uniform memory access or NUMA); moreover, some processing elements (hardware accelerators or DSP) are frequently constrained to read their input data and store their output results in specific memories.

Programming a baseband processor is a difficult task because of multiple reasons like heterogeneity of computing and control units, communication channels or memories, real time constraints, increasing dynamicity of communication standards, simultaneous coexistence of multiple radio technologies, etc.

On this kind of architecture, the development of functionally correct applications, satisfying the performance constraints, requires a deep knowledge of the target platform. The development time is thus significantly increased, compared to more classical computer architectures. A possible solution to this kind of difficulties is the automatic generation of the software application code from a high abstraction level model of the application, the target architecture and the communication patterns. The detailed knowledge of the target architecture is thus required only in the architecture modeling phase. The architecture model is reusable for all application developments on the same platform and designers can focus on the modeling of the application.

This paper shows how application models can be mapped onto an architecture and then transformed into an executable software code which can run on the target hardware platform. The paper is structured as follows: section 2 presents the modeling methodology. Section 3 shows how the hardware platform is abstracted through a collection of software components usable by the code generator. Section 4 uses Embb, an experimental hardware platform developed at Telecom-ParisTech, as an example target architecture, to show the hardware abstraction in action and how it simplifies the code generation.

2 Modeling

This section presents the Ψ-chart modeling methodology the code generation relies on. This methodology targets *data flow* applications, that is, applications where operations are performed on continuously flowing data. This is typically the case in signal processing application. This methodology is the result of several previous research works done at Telecom-ParisTech.

The Ψ-chart methodology is a modeling technique for embedded systems in which the models of the hardware architecture, the application and the communication pattern are designed independently. This separation allows the simplification of the modeling phase of each part, allowing to design a model without using details from other models.
The 3 types of models of the Ψ-chart methodology are:

- The application model, a network of interconnected processing blocs representing the different operations to be executed on the data. The processing blocs are composed of a part dedicated to the data processing (the processing algorithm) and a part dedicated to the control (the part that fires the execution of an operation).
- The hardware architecture representing the physical system on which the application is executed: processors, hardware accelerator, memories, DMA controller, buses, etc. For illustration, we mainly use Embb in our platform models. It is an architecture for software defined radio [1], developed by our team (Telecom-ParisTech, LabSoC). However, the work presented in this paper can still be applied on other architectures.
- The communication pattern model describe the different protocols to exchange data between elements of the physical system [2]. In the case of Embb, we can show for example how data can be exchanged between the internal memories of two DSP.

The methodology consists of the following steps:

1. Creating the application model, independently of the architecture or the communication patterns
2. Creating the architecture model, independently of the application or the communication patterns
3. Creating the communication patterns, independently of the application or the architecture
4. Mapping phase:
   (a) Each task described in the application model is mapped onto an element of the architecture which can execute this task.
   (b) For each data transfer described in the application model, a communication pattern is selected and mapped to the hardware elements that will implement it.
5. The performance requirements (timing constraints, energy consumption, etc.) of the resulting model are checked.
6. If some requirements are not fulfilled, the models are reworked, from the simplest changes to the most complex. Usually, other mappings are first explored, followed, if needed, by the addition of new communication patterns, because these modifications do not imply changes of the application or architecture

models. If requirements are still not satisfied, the application model can also be modified. Finally, if it is still not sufficient, and if it is possible, the architecture model is reworked to include more resources. Of course, depending on specific constraints, this order can be different.

These steps are repeated until the final model fulfills all system requirements. The input of the code generator is the final model obtained after the mapping step of the Ψ-chart methodology. In this model, each operation is associated to a computing unit and each data transfer to a communication pattern. In our proposal for software code generation, each element of the mapped model has a software component counterpart that the code generator assembles to produce the software application. The next section details these software building blocks.

3 Software Components

This section presents the software components used as building blocks by our proposal of code generator. They are key elements of the hardware platform abstraction. They are meant to be generic enough, such that their Application Programming Interface (API) can be reused on every hardware platform. The software components fall in two categories: i) the operations related component and ii) the runtime environment components. In the following we will describe their role during the execution of the generated code.

3.1 Operation Related Components

The operation related components are associated with one operation block in the application model mapped onto a processing unit in the architecture model. They consist of a list of functions and data structures which will be used by the code generator to create the executable code. Thus, we need to provide this set of function/data structure for each pair of operation/processing unit possible in the architecture. The set consists of 2 data structures and 4 functions. The data structures are:

- the parameter data structure: contains all the parameters required by the processing unit to execute the operation
- the operation data structure: contains the parameter data structure, as well as scheduling information such as the operation identifier, the operation deadline, the operation priority and the operation dependencies.

The functions are:

- the init function: called once at the beginning of the program. It takes the operation data structure as parameter.
- the parameter function: called each time the operation needs to be executed. Its parameters are: the operation data structure, the addresses of the input and output data, and the all the parameters from the model.
- the start function: called after the parameter function. It takes the operation data structure as parameter.
- the exit function: called at the end of the program.

To generate the code of the application, the generator first search for every pair of operation/processing unit in the mapped model. For each found pair, it instantiate the corresponding operation data structure. After the data structure instantiation, the init function is called, once for every pair of operation/processing unit found. This initialisation phase is also done for every pair of transfer operation/transfer unit in the application model. Once the initialisation phase is done, the generator will go through all the operations that needs to be executed. For each operation, it will first called the allocator, a runtime environment component described later, of the memory in which the data has to be stored (for example the local memory of a DSP unit or the global memory for a general purpose processor). According to the dependencies of the operation, the allocator can be called for two different purpose:

- 1) the input data are generated by another operation in the same memory. Then the allocator is called to get the address of the input data, and to allocate a memory space for the output data.
- 2) the input data are generated in another memory. Then the allocator is called to allocate memory for both the input and the output data.

The parameter function of the operation is called, and those addresses are given in the parameters, along with the operation data structure and the parameters of the application model. When the parameters of the operation are set, the start function is called. This function will transmit the operation request to the scheduler of the processing unit. The scheduler is a runtime environment component which will be described later in this section. If the input data are generated in another memory, then a data transfer has been programmed in the application model. The parameter function of this transfer is called, the output address of the transfer is the address of the input data just allocated for the operation, and the input address is the output data address of the operation that has generated the data. The start function of the transfer is then called. At the end of the code, for each pair of operation/processing unit and transfer operation/transfer unit, the exit function is called.

3.2 Runtime environment component

Unlike the operation related components, the runtime environment components are not a set of function and data structure used by the code generator, but a set of functionalities that must be on the target platform and which will be called by the executable code of the application. There are two major categories of runtime environment components: the scheduler and the allocator.

The scheduler component is associated to a processing unit or a transfer unit. It receives operation or transfer request from the application, selects the next operation or transfer to be executed according to the scheduling information and starts the operation or the transfer. The scheduler receives the requests thanks to the start function of the operation related component category. The start function actually send the operation data structure to the scheduler. This data structure contains the scheduling information for request selection, and the parameter data structure. This means that the scheduler must be able to actually start the real operation using only the information in the parameter data structure. The scheduler must also select the new task to execute. For this purpose, it needs to have a scheduling algorithm, and a mechanism to update the data dependencies. This mechanism can be a function which check if a given task with a given identifier has been completed or not. This function must be callable by the other runtime environment component. For example, if an operation on a given processing unit cannot start before a data transfer has been completed by a given transfer unit, the scheduler of the processing unit must be able to ask the scheduler of the transfer unit for the completion state of the transfer.

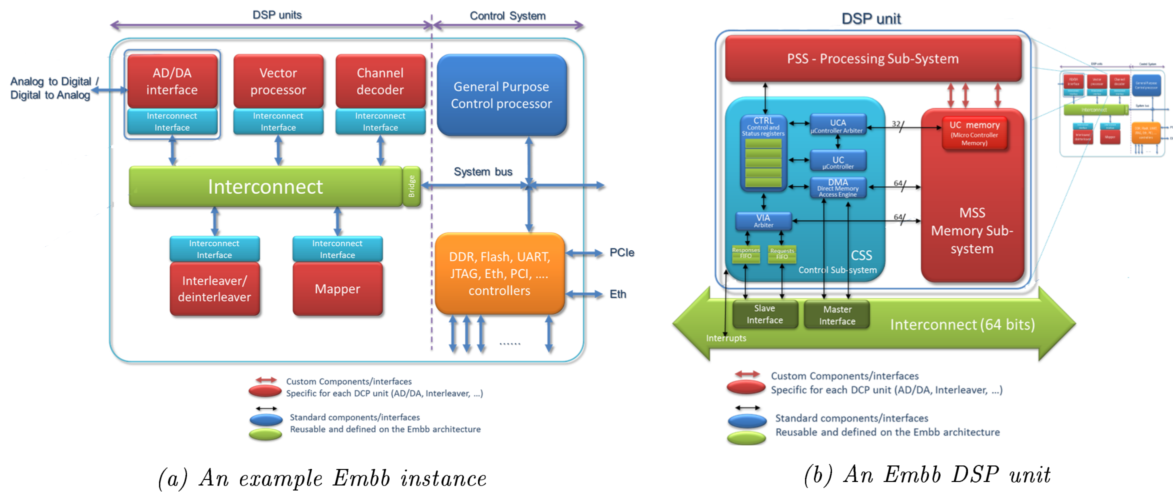
The memory component is associated to the local memory of a processing unit, or the global memory. Its purpose is to provides free addresses of its memory to the application, and to keep track of all the memory allocated. The allocation request takes the size of the data as a parameter, as well as information on how long this address space must be kept. This information consists of the task which needs the data, and the number of times the task can be executed before the data is freed. Thus, the data can be allocated for only one task execution, several, or until it is manually freed by the application. The allocator will use the same mechanism as the scheduler to know if the task are executed and the data can be freed.

4 Use case: Embb

This section illustrates how the software components described in the previous section have been implemented to abstract the Embb[1] hardware platform. After a brief description of the Embb platform, we will present two operation related components, one for a computing task and one for a data transfer. We will also present the actual state of the runtime environment component for one of the processing unit of Embb. We finish by presenting a small code written using the components presented in the previous section

4.1 Embb

Embb is a generic architecture for Digital Signal Processing. An instance of the Embb platform (figure 1a) consists of a cluster of digital signal processing units (DSP) interconnected with a crossbar. The DSP cluster is connected to a general purpose processor and a global memory via the crossbar. In our example use case, the general purpose processor runs a GNU/Linux distribution, and accesses the DSP units through Linux device drivers.
The Embb DSP units (figure 1b) are specialized for a type of processing (vector processing, interleaving, mapping, etc.) Each DSP unit combines a processing subsystem, a memory subsystem and a control subsystem. The processing subsystem is the core of the DSP unit, where all the computations are done. The control subsystem consists of a micro controller, a direct memory address engine (DMA) and a set of interface registers used to control the DSP unit. The memory subsystem is the local memory of the DSP unit. The interface registers and the memory subsystem are mapped in the local micro controller address space and in the global address space of the general purpose processor. The local micro controller and the general purpose processor can



(a) An example Embb instance

(b) An Embb DSP unit

Figure 1 – The Embb generic architecture

thus directly access them to set processing parameters, launch processing, launch DMA-driven data transfers, read exit statuses... In normal operations the input and output data are moved around using the DMA engines but, as the memory subsystem is mapped in their respective address space, the local micro controller and the general purpose processor can also access the data through regular load / store operations.
The software components must initialize, set the parameters and control the processing units to execute the operations described in the application model. It is necessary to know how the target architecture works to be able to implement those components. The DPS of Embb all works the same way:

- Access to the DSPs must me authorized by writing in some control register in the crossbar.
- DSP must me activated by writing in some control register in their own control subsystem.

To launch an operation in a DSP, all the following steps must be done:

1. Set the parameters of the chosen operation. All the DSPs have a set of register in their control subsystem which allows to modify the operation parameters. The number of parameter register can change depending on the DSP, but their local address (in the DSP address space) is the same for all DSPs.
2. Once the parameters set, the operation is fired by writing in one of the register of the control subsystem (the status register). The address of this register is the same for all DSPs (in the DSP address space).
3. When the operation is done, an interrupt is triggered. A second operation can be launched.

The DSPs of Embb have a hardware FIFO mechanism. It allows to set the parameter of another operation while one is still executing, such as the new operation is automatically started when the first is finished. This mechanism increases the DSP efficiency.

4.2 Example of components

We first take the example of the Fast Fourier Transform (FFT) operation. In the case of Embb, this operation is commonly executed by a DSP unit called the Front End Processor (FEP), which is specialized in vector computing. Creating operation related component for this pair of operation/processing unit will allow us to include FFT bloc in our application model, and to map them onto the FEP bloc of the architecture model, resulting in a mapping model which can be used to generate a code executable on Embb. The first component to be designed is the parameter data structure. This data structure must be designed so it is enough to launch an operation on the processing unit. As we presented previously, to start an operation on a DSP, we first need to write the parameters of the operation in some control register. Our parameter data structure has thus a field representing each of the parameter register of the DSP (since the DSPs have three parameters register at most, we gave our data structure three fields). The scheduler component will only have to copy the content of those fields in the parameter registers of the DSP to properly configure the operation. The second data structure needed is the operation data structure. It contains the previously described parameter data structure, as well as the operation identifier, its deadline, its priority and its dependencies. Those dependencies are represented by the list of the identifier of the operation that must be completed for the current one to be executable.

Once the data structure are defined, we have to implement the initialization function, the set_parameter function, the start function and the exit function. In our case, the initialization function is very simple. Since we access the DSPs with device drivers, the main initialization part is done when the hardware platform is booted, and not during the execution of the application. This is the reason why our initialization functions contains no code to initialize the DSP. However, we use those functions to set some parameters in the parameter data structure. We are doing this when the parameter value is completely determined when the operation type is known. For example, the parameter that set the operation type would be set to "FFT". The second function is the set_parameter one. This function will simply copy the missing parameters in the data structure. Those parameters can be the addresses of the input and output data, if the FFT is an inverted one, etc. It will also complete the other fields of the operation data structure with information taken from the model (identifier, deadline, priority and dependencies). The start function has to transfer the operation data structure to the scheduler component. In our case, the scheduler component is integrated in the DSP driver. This transfer is done with a mechanism called ioctl. From the implementation perspective, we simply call this ioctl in the start operation and we give it the operation data structure as a parameter. In our version of the software component, the exit function is empty. All of the exit operation are done by the drivers when shutting down the hardware platform.

The data transfer operations work the same way as the computing operation in the case of Embb DSP. The DMA controllers of each DSP are configured the same way as the processing subsystems, we implemented the transfer operation components the same way as the computing operation component.

At the opposite of the operation related components presented above, which can be described as data structure and function libraries used by the code generator to produce the application code, the runtime environment component are services on which the application code relies. We decided to adapt the device drivers of Embb so that they provide those services. The drivers of Embb are the implementation of the runtime environment in our case. They are implemented in a generic way. There is only one driver, instantiated for each DSP in the Embb architecture. We will present how this driver provides the services required to execute the generated code. The scheduler must provide two main services: receive the operation execution requests, and signal the end of execution of one operation. Those services are called using the ioctl mechanism described above. One ioctl is used to add an operation in the waiting list of the scheduler, another one is used to get the completion state of an operation. Currently there is no scheduling algorithm in the scheduler. It will only execute the operation in the same order they were requested, without checking dependencies. However, it is possible to request the execution of an operation, and to retrieve the completion state of an operation, so the dependency checking can be done manually in the application code. The allocator does not currently exists. The memory locations must be manually provided.

5 Conclusion

In this work we presented a methodology to abstract the hardware platform, so that we can automatically generate a code that can be run on it from a high level model. This methodology relies on the definition of software components usable by the code generator (the operation related component) and the generated application (the runtime environment component). The software components have been designed regardless of the target hardware platform, but with the source model in mind. The consequence of this approach is that those component can be implemented for a large number of possible target platform. They can also be used to generate code for simulation purpose. The future work will be to complete the runtime environment component by adding a scheduling algorithm to the scheduler, and by implementing a basic allocator. The code generator must also be modify to use those component to generate the code, so that we can run basic generation tests.

6 References

[1] R. Pacalet, “Embb, a generic hardware and software architecture for digital signal processing (`http://embb.telecom-paristech.fr/`),” 2016.

[2] L. Apvrille, A. Enrici, and R. Pacalet, “A UML Model-Driven Approach to Efficiently Allocate Complex Communication Schemes,” in *MODELS 2014*, (Valencia, Spain), Sept. 2014.