Multi-Device Driver Synthesis Flow for Heterogeneous Hierarchical Systems

Alexandre Chagoya-Garzon
Aselta Nanographics, Grenoble, France
Alexandre.Chagoya-Garzon@aselta.com

Frédéric Rousseau, Frédéric Pétrot
TIMA laboratory (CNRS/UJF/Grenoble INP), Grenoble, France
Frederic.Rousseau@imag.fr, Frederic.Petrot@imag.fr

Abstract—Heterogeneous hierarchical architectures result from the interconnection of several heterogeneous MPSoCs through an efficient communication infrastructure. Each communication between two processing units requires the use of hardware devices managed by a software driver responsible to initialize, handle and complete the communication.

This paper describes a multi-device driver synthesis flow based on available communication paths of the architecture. A small set of generic driver templates is available in a driver library. The flow is in charge to select one correct driver template from the library, and then to configure and specialize it in order to produce the source code. The effectiveness of our approach is illustrated by a significant example.

Index Terms—Communication drivers, heterogeneous MPSoCs, software synthesis

I. INTRODUCTION

System-on-Chips (SoC) are widely available and combine the heterogeneity of their processors, usually GPP and DSP, with efficient and specific I/O infrastructure and a non-standard on-chip memory organisation and access, at a low silicon area and power consumption cost. We call this SoC a tile in the rest of this paper. Typical examples of tiles of the hierarchical architectures we consider are the OMAP [1], Nexperia [2] or Diopsis940 [3] platforms. Scaling such an architecture to 100 or more computational units is a challenge for application programmers, and some tools need to be introduced to hide the complexity of the whole computation and communication infrastructures, even if hiding it totally is an open issue and seems overall impractical in the general case.

Efficient use of the communication infrastructure is one of the main challenges in this context. Indeed, the above mentioned architectures are heterogeneous in terms of processing elements, hierarchical including three levels as multiprocessor tile, multi-tile on a chip, and multi-chip. However, the application programmer must not suffer from this complexity. To make this possible, communication synthesis approaches and tools should be provided.

Traditionally, software generation flows (Y-chart based [4][5]) start from high-level models of the application, architecture and mapping. At this abstraction level, the mapping concerns tasks and communication channels used for the application model. The task mapping is the binding between tasks of the application and the hardware resources providing communication services in the architecture. Communication depends on the task mapping, and thus on the devices involved in the data transfer in the communication paths between the two processing units.

Usually, the way data transfers occur in the architecture is predefined and implicit. Due to the multiple levels of hierarchies, describing explicitly in the architecture model the different stages of data transfers becomes a necessity. A simple way to do it is to enumerate all communication paths between two processing units, and for each communication path all devices involved in the transfer.

A driver is a software program used to interact with hardware devices. Writing one driver per communication path is not conceivable anymore due to the large number of available communication paths in a complex architecture. One solution is to develop generic driver templates that implement several communication paths. For each communication path, a correct template is selected and specialized to efficiently handle the hardware resources. Thus, we obtain a multi-device driver that is then compiled and linked with the operating system and application code to provide the binary code for each processing unit. This work focuses on the multi-device driver synthesis flow and the use of a formalization of communication paths. We provide a communication synthesis process with a good simplicity/control trade-off.

Section 2 gives an overview of the heterogeneous hierarchical systems and details the requirements for the driver synthesis. Section 3 deals with the formalization of communications in such architectures. Section 4 explains the use of hardware communication paths. The multi-device synthesis flow is detailed in section 5. Through an experiment, section 6 illustrates the effectiveness of our approach.

II. HETEROGENEOUS SYSTEMS

A. Overview

The systems we are focusing on are heterogeneous because the tile is heterogeneous (processors, memory hierarchy, I/O organization), or of different kinds. The whole interconnection infrastructure is not imposed, however we can distinguish three interesting hierarchy levels:

- At tile level, a local interconnect with bus or crossbar
- At chip level, connection of different tiles through a NoC (Network On Chip)
• At global system level, connection of several chips through a network (3D Mesh, high performance bus, ...).

An overview of hierarchical systems is presented in Figure 1. Three kinds of blocks can be found in this figure: the CPU and their local subsystem, the memories, and Network Interfaces which allow the tiles to communicate. Memories do not have a uniform access cost (hierarchical architectures are thus NUMA [6]), or uniform access method when DMA is used to access some memory devices.

We assume the presence of a special peripheral attached to each tile of the system and making the bridge with the outside communication fabric. In addition to usual compiling and linking toolchains, it is also indispensable to have an Operating System (OS) to manage multi-tasking and I/O accesses. To get the binary code for the whole architecture, the last step is the communication synthesis, based on device drivers.

![Figure 1. Hierarchical system illustration](image)

**B. Challenges on communication synthesis**

A software synthesis approach consists in providing a specific software component tailored and called by other software components. This facilitates the use of specific hardware capabilities and the best communication path, depending on the context (data size, performance, protocol, ...).

The application programmer must be able to write his tasks and their data exchanges without caring in a first phase about the underlying middleware, tile and global hierarchical system with its communication infrastructure. This transparent handling of communications must leave him a certain level of control on the way communications will be implemented. The choice of the data transfer mechanism has to be done once the tasks have been assigned to a computing unit.

Communication synthesis has been a field of research in co-design, specifically targeting the co-design of an hardware IP and its controlled software suited for a given communication [7][8][9][10]. [9] gives a solution for generating a device driver complying with IP communication protocol and the CPU. The methodology is based on the extraction of the formal model of the IP communication protocol. Then a driver template is selected and customized to generate the device driver compliant with the CPU and the operating system. The starting point of the methodology is the RTL IP-core and the related testbench, which unfortunately does not fit with third party devices. [11] provides a solution to generate a device driver from device features model and specifications of the hardware device and operating system interfaces. This is a promising solution but limited to single device drivers.

There are some other works that are still in the research domain for FPGA targets in the context of high-level synthesis [12][13], as the current high performance architectures cannot be enhanced anymore by ad-hoc specific communication IPs for cost reasons, or for MPSoC with the synthesis of a specific communication interface between two processing units [10] and the corresponding drivers. Unfortunately, this does not fit with the platform-based context as we target existing platforms.

[14] tackles the problem of the inefficient implementation of synchronous models. A method is proposed to implement a synchronous model on a Loosely Time-Triggered Architecture (LTTA) while preserving the equivalence in terms of communication.

Our software synthesis approach relies on multi-device driver templates. This multi-device driver template is a generic software element, which needs to be specialized to drive all the hardware resources involved in a specific communication. Depending on many parameters (endianness, wordwidth, processor architecture like GPP or DSP, ...) and on the communication path, the template is selected and tailored to produce the source code.

**C. Hardware communication paths**

In a given mapping, each communication channel of the application is associated to a list of hardware resources involved in the data transfer. This list has led us to introduce the hardware communication paths in the architecture model. We define a path as a list of hardware resources starting from an accessible memory of a processing unit and ending on a communication media (a memory or a peripheral). There exist several paths between a CPU and a media. Figure 2 shows a possible communication organization inside of a given tile. The arrows symbolize the hardware resources and memories involved during a read or write communication process over this path. A CPU triggers the communication, but it can be helped by specific hardware resources (DMAs). The hardware communication paths are symbolized by arrows taking a local memory of a CPU as source or target of a communication, another memory of the system serves as exchange media in this case.

![Figure 2. Hardware communication paths inside a tile](image)

We can make some observations on this simple and intuitive representation. Firstly, the communication mechanisms are well suited to support a message-passing paradigm. A
second remark is that communication paths are point-to-point, unidirectional channels which fit with a process network representation (KPNs for example). A last remark is the asymmetry of the communication paths. Hardware accelerators like DMAs may be used to optimize the access to some networks. This has a consequence on the binary code, as the access to a given hardware resource will not be done in the same way according to the target CPU and hardware communication path. One solution to make this simple is to rely upon a Hardware Abstraction Layer (HAL). This means that the binary generation flow imposes the existence of a HAL that interacts with all the hardware components that belong to a specific communication path.

From the previous remarks, one can notice that a multitude of hardware paths may exist within a tile, as well as for inter-tile and inter-chip communication, even if a specific peripheral or network interface facilitates the access to the network.

Therefore, we need a well-defined formalism to introduce all the hardware paths of a hierarchical system.

III. HARDWARE COMMUNICATION PATHS FORMALIZATION

A. Intra-tile communication paths

Let \( T \) be the set of tiles, and \( T \) an element of this set. \( T \) is defined as follows:

\[
T = \{ \mathcal{B}_T, \mathcal{I}_T, \mathcal{L}_T \}
\]

where \( \mathcal{B}_T = \{ \mathcal{PU}_T, \mathcal{M}_T \} \) and \( \mathcal{I}_T = \{ \mathcal{N}_T, \mathcal{N}_T \} \).

\( \mathcal{B}_T \) is the set of processing units (\( \mathcal{PU}_T \)) and memories (\( \mathcal{M}_T \)) of the tile. \( \mathcal{I}_T \) is the communication infrastructure of the tile composed of communication networks (\( \mathcal{N}_T \)) and of different network interfaces (\( \mathcal{N}_T \)). \( \mathcal{L}_T \) represents the set of links between a hardware block of \( \mathcal{B}_T \) and a network of \( \mathcal{N}_T \) and of composed of (\( \mathcal{Block}, \mathcal{Net} \)) or (\( \mathcal{Net}, \mathcal{Block} \)) pairs where \( \mathcal{Block} \in \mathcal{B}_T \).

We define two sets for the intra-tile communication paths between a computing unit noted \( \mathcal{PU} \) and a memory noted \( \mathcal{Mem} \): the set of write and read hardware communication paths. We adopt the "." sign as the mathematical concatenation operator between two architectural elements, and \( (E)^* \) as the mathematical concatenation of a given number of \( E \) elements (0 or n elements).

The write communication paths definition are:

\[
\text{WrPaths}_T(\mathcal{PU}, \mathcal{Mem}) := \{ \text{Memo}_0 \text{Net}_0 \cdot (\mathcal{NI}_0 \cdot \text{Net}_{i+1})^* \cdot \text{Mem} \}
\]

where \( \{ \text{Memo}_0, \text{Net}_0 \}, (\text{Net}_{last}, \text{Mem}) \in \mathcal{L}_T \)

The read communication paths definition is similar:

\[
\text{RdPaths}_T(\mathcal{PU}, \mathcal{Mem}) := \{ \text{Memo}_0 \text{Net}_0 \cdot (\mathcal{NI}_0 \cdot \text{Net}_{i+1})^* \cdot \text{Mem} \}
\]

Put in another way, an intra-tile write (resp. read) communication path binds a CPU with its local memory to another accessible memory used as communication media, which represents the storage of emitted (resp. received) messages. The communication path uses the communication infrastructure of the tile, composed of networks and network interfaces, to connect this memory to the \( \text{Mem} \) memory for a write (resp. read) access.

The following example gives the write communication path between the RISC memory to the external memory (based on Figure 2) through the different bridges and buses:

\[
\text{WrPath}(\text{RISC, ExtMem}) = \{ \text{RiscMem} \cdot \text{RiscBus} \cdot \text{RiscBRIDGE} \cdot \text{SystemBus} \cdot (\text{BRIDGE} \cdot \text{ExtMemBus} \cdot \text{ExtMem}) \}
\]

We can now introduce the set of read and write communication paths internal to a tile \( T \): \( \text{WrIntraPaths}_T \) and \( \text{RdIntraPaths}_T \), as the set of all write (resp. read) hardware communication paths between each CPU and each shared memory of the tile (see Definition 8 below).

B. Inter-tile communication paths

Let \( \mathcal{H} \) represent the set of hierarchical systems, and \( H(n) \) a n-tile system. \( H(n) \) is defined as follows:

\[
H(n) = \{ T^n, \mathcal{I}_H(n) \}
\]

where \( \mathcal{I}_H(n) = \{ \mathcal{N}_H(n), \mathcal{N}_H(n), \mathcal{L}_H(n) \} \)

A hierarchical system is composed of several tiles \( \{ T_1, T_2, \cdots, T_n \} \), and thus of a set of CPUs and memories that we will note \( \mathcal{PU}_H(n) \) and \( \mathcal{M}_H(n) \):

\[
\mathcal{PU}_H(n) = \bigcup_{i=1}^n (\mathcal{PU}_{T_i}) \quad \text{and} \quad \mathcal{M}_H(n) = \bigcup_{i=1}^n (\mathcal{M}_{T_i})
\]

The tiles are connected through an inter-tile communication infrastructure \( \mathcal{I}_H(n) \). This infrastructure set is in turn constituted of a set of networks (\( \mathcal{N}_H(n) \)), network interfaces (\( \mathcal{N}_H(n) \)) and of \( \mathcal{L}_H(n) \), composed of oriented pairs (\( \mathcal{NI}, \mathcal{Net}, (\mathcal{Net}, \mathcal{NI}) \)) representing the connections between those first two sets. The hierarchical system network interfaces are included in the union of all tiles network interfaces:

\[
\mathcal{N}_H(n) \subset \bigcup_{i=1}^n \mathcal{N}_T_i
\]

In the same way as for the tile, we define the inter-tile communication paths, however between a processing unit \( \mathcal{CU}_{T_i} \in T_i \) and an inter-tile network \( \mathcal{Net}_H \in \mathcal{N}_H \). The set of inter-tile write communication paths is defined as follows:

\[
\text{WrPaths}_H(\mathcal{CU}, \mathcal{Net}_H) := \{ \text{Memo}_0 \text{Net}_0 (\mathcal{NI}_0 \cdot \text{Net}_{i+1})^* \cdot \mathcal{M}_H \cdot \mathcal{Net}_H \}
\]

where \( \{ \text{Memo}_0, \text{Net}_0 \} \in \mathcal{L}_{T_i} \)

The set of inter-tile read communication paths is similar:

\[
\text{RdPaths}_H(\mathcal{CU}, \mathcal{Net}_H) := \{ \text{Memo}_0 \text{Net}_0 (\mathcal{NI}_0 \cdot \text{Net}_{i+1})^* \cdot \mathcal{M}_H \cdot \mathcal{Net}_H \}
\]

In other terms, a write (resp. read) inter-tile communication path represents the communication infrastructure of the tile needed to connect a CPU (bound to a local memory \( \text{Memo}_0 \)) to a network interface connected to an off-tile network for a write (resp. read) operation. An example of Read and Write communication path for inter-tile is given in the Experimentation section.
A hierarchical system disposes of a set of read and write inter-tile communication paths ($\text{WrIntPaths}$ and $\text{RdIntPaths}$) which connect every CPU of the tile to each off-tile network of the system (Definition 9). We define the set of communication paths of the whole system, $\text{HubComWrPaths}_H$ and $\text{HubComRdPaths}_H$, each one being the union of all intra-tile and inter-tile write (resp. read) communication paths (Definition 10).

$$\text{Intrapaths}_T = \{\text{WrIntraPaths}_T, \text{RdIntraPaths}_T\}$$ (8)

where

$$\text{WrIntraPaths}_T = \{\text{WrPaths}(\text{CPU, Mem}) / (\text{CPU, Mem}) \in P_T \times M_T\}$$

$$\text{RdIntraPaths}_T = \{\text{RdPaths}(\text{CPU, Mem}) / (\text{CPU, Mem}) \in P_T \times M_T\}$$

$$\text{InterPaths}_H = \{\text{WrInterPaths}_H, \text{RdInterPaths}_H\}$$ (9)

where

$$\text{WrInterPaths}_H = \{\text{WrPaths}(\text{UC, Net}) / (\text{UC, Net}) \in U_H \times N_H\}$$

$$\text{RdInterPaths}_H = \{\text{RdPaths}(\text{UC, Net}) / (\text{UC, Net}) \in U_H \times N_H\}$$

$$\text{HubComPaths}_H = \{\text{HubComWrPaths}_H, \text{HubComRdPaths}_H\}$$ (10)

where

$$\text{HubComWrPaths}_H = \bigcup_{T \in L_H} \text{WrIntraPaths}_T \cup \text{WrInterPaths}_H$$

$$\text{HubComRdPaths}_H = \bigcup_{T \in L_H} \text{RdIntraPaths}_T \cup \text{RdInterPaths}_H$$

IV. HW COMMUNICATION PATHS AND Y-CHART MODEL

A. Application model

We choose a simple process network model for the application. We only show the process network structure, and do not specify here the functionality of each process. The only requirement for this process implementation is the exclusive use of a communication library for inter-process communication (read/write, send/recv, ...). We introduce an application $A$ as a set:

$$A = \{\text{Process}_A, \text{Channels}_A, \mathcal{L}_A\}$$

Connections are oriented pairs, following relation is verified in the $\mathcal{L}_A$ link set:

$$\forall C \in \text{Channels}_A, \exists!(P_0, P_1) \in \text{Process}_A / ((P_0, C) \in \mathcal{L}_A) \land ((C, P_1) \in \mathcal{L}_A)$$ (11)

In other terms, software channels are point-to-point and unidirectional.

B. Mapping

Finally, the mapping of application elements onto a multi-tile architecture is composed of two functions: task_mapping maps each process of the application model to a computing unit of the system;

$$\text{task\_mapping} : \text{Process}_A \mapsto \mathcal{P}_U_H$$

channel_mapping maps each channel of the application onto a read and a write communication paths of the system.

$$\text{channel\_mapping} : \text{Channels}_A \mapsto (\text{HubComRdPaths} \times \text{HubComWrPaths})$$

The channel_mapping function applied to the channel $C$ defines a write and read communication paths:

$$\text{channel\_mapping}(C) = (WP, RP)$$ (12)

$$(WP, RP)$$ are intra-tile communication paths for a channel connecting two processes belonging to the same tile (through a shared memory $\text{Mem} \in M_T$), and inter-tile communication paths for two processes belonging to different tiles (through an inter-tile network $\text{Net} \in N_H(n_i)$).

C. Use of hardware communication path

We assume that at least one (correct) hardware communication path is given for communication between two processing units. We assume also that all hardware communication paths are listed in the architecture model and respect the previously given formalization. Same for the mapping which is supposed to follow the rules mentioned in the previous section.

Two main reasons lead us to the need of hardware communication paths. The first one is the mapping process that is usually done by a system designer who is not the hardware architecture designer. Such an information of available communication paths helps to map a communication channel with the most efficient communication path.

The second reason is the multi-device driver synthesis which is basically a composition of drivers with care of chaining, dependencies and parametrization. Indeed, all the hardware components involved in the communication path for the data transfer are supposed to be initialized and managed by a software component running on a processor. Moreover, in a driver library, all drivers should contain an information of the possible communication paths they drive in order to facilitate the automatic selection of valid drivers.

V. MULTI-DEVICE DRIVER SYNTHESIS

A. Layered binary stack

For each processor of the system, we adopt a layered binary structure similar to that found in other existing operating systems of the MPSoC world like eCos [15]. Application tasks are implemented as threads of the chosen operating system (OS). Task behaviour is coded by the user and is associated to the application model. It uses a message-passing library to communicate with the other threads of the application. The implementation of communications is thus transparent for the application programmer, channelRead and channelWrite are in this case the only available communication primitives. The middle layer is composed of an OS and the communication interface implementation. It accesses the hardware resources exclusively through the HAL (Hardware Abstraction Layer) API, which is inspired by the one used in eCos. The HAL encloses a minimal set of low-level software related to the CPU (context switch, I/O operations, interrupts) and platform (endianness management, hardware locks). An important point to highlight is that the HAL allows an optimal and fixed access to all the elements of the tile.

B. Generation flow overview

Figure 3 presents the adopted generation flow model, composed of a front-end part which parses the abstract models (under an XML format), together with the task implementation code, to prepare the software components for each processor
and a back-end part which compiles and links these elements to obtain a binary code.

The front-end part of the flow relies on two modules. A code generation module generates the necessary initializations and launches the user-written application tasks in the threading system of the operating system. The second module is the component selection module. This module selects the software components for the OS, communication and HAL that are added to the behavioral task code and initializations. This module takes as input information from the high level models (application, architecture and mapping) as well as some information from the system designer. For example, the list of OS services requested by tasks running onto a processor can be automatically determined.

The OS disposes of a set of drivers, each one implements a given message-passing communication mechanism, based on a shared memory or a network.

D. Driver library

The basis of the communication synthesis is thus the driver library. Writing a driver for such a flow is not an easy task for several reasons. First, writing drivers is a difficult and error-prone task in reason of the number of interfaces that they have to respect (OS, device, platform). Moreover, in front of the number of communication paths, it is not conceivable to write one driver for each communication path of the system.

We have chosen to write drivers based on templates. Each template offers a certain degree of reconfigurability in order to be eligible for a given set of communication paths. Thus, for each channel mapped on a communication path, the flow is in charge of configuring the drivers to meet the high level specification. This is not an easy task, because some parameters (like memory addresses) can only be fixed at link time, thus very deep in the flow. Finally, the driver relies on a HAL fixed by the OS. The HAL proposes an optimal way to reach the close environment of the CPU, it is thus hard and not always wise to bypass this implementation. The difficulty of writing drivers for this flow has made us choose a compromise between simplicity and efficiency on one side, and respect of the abstract model on the other side.

The communication synthesis in our flow model consists thus in a selection of a driver template for each communication channel attached to a communication path, and a specialization of this driver to configure all the settings necessary to make it work and respect the high-level model specification (memory addresses, tile rank, channel id, ...).

E. Driver selection

The driver selection consists in choosing the most appropriate driver template in the abovementioned driver library. A driver implements a channel of the application model and it has to manage and drive all hardware components corresponding to the selected communication path. The mapping file contains several informations about two bound tasks: the information whether the communication is of intra or inter tile nature, as well as the selected communication path which indicates which memory (or peripheral) serves as communication media, and thus the list of devices to drive. A driver fits with the communication path if one of the list of peripherals it drives is the same as the one of the selected communication path. This correspondence supposes to have a consistency between these two lists, and this is made possible thanks to the presented formalization. If only one driver fits, it is selected. If several drivers fit, the system designer has to choose the most appropriate. This choice depends of any cost function (performance, ...) and on the protocol that fits. Indeed a driver implements at least one protocol, but it could also implement several protocols and their use may depend on the context (size of data to transmit, ...). The matching between the communication path and the list of hardware resources managed by a driver could be done by an automatic tool to look for all possible drivers, as well as the automatic choice among several candidates. Finally, the selected driver has to be specialized.

C. OS model and communication drivers

Our binary generation flow relies on an OS with a classical microkernel structure [16][17]. The OS uses a minimal HAL layer to access the hardware. It is composed of userspace modules (POSIX threads, files, drivers) and kernel-space modules with a core, a boot module and a Virtual File System (VFS). We choose a modular architecture, where basic modules are easily interchangeable at compile and/or link time.

Each communication channel of the application model is associated to a virtual file of the OS. The VFS is an uniform way offered by the OS to the application to access any kind of files, including devices or networks. It offers a standard interface to the communication layer, which can be seen as a wrapper of the VFS and masks the underlying communication implementation to the application programmer. Each virtual file is associated to a driver in a generated initialization file.
F. Driver specialization

The specialization consists in giving specific low level information required in the driver source code before compilation. Two kinds of information remain to be set up. The first one regards the platform (or architecture) dependency. Addresses of peripherals, type and related addresses for interrupts, as well as for locks, and some others like memory map for FIFO instance should be given. Some of them may be described in the architecture model, but these details are usually not inserted in a high level model of the architecture. In such a case, they have to be added manually by the system designer, or extracted from a specific memory map description file.

The second kind of information concerns the driver kernel itself. As explained in a previous subsection, the access to the communication path is done by a write and read in a virtual file. The name of the virtual file is decided during the initialization file generation based on the information from the application, architecture and mapping files. Indeed, this virtual file is shared between the read and the write associated drivers for a communication. If the driver includes several different protocols, the specific part has to be validated. The size of the required internal buffer is allocated when the protocol has been decided.

A configuration tool helps us specializing these drivers [18]. The input architecture model does not contain enough information to fill all hardware-related values which can be obtained with a more detailed description (physical hardware addresses), computed with default values or given by the system designer.

VI. EXPERIMENTATION AND DISCUSSION

A. Target multi-tile architecture and Application

The target multi-tile platform is based on the Atmel Dioprisis940 [3], composed of an ARM9 subsystem, a DSP (Atmel’s mAgicV) subsystem, a set of peripherals and an external memory interface. All these blocks are connected through a layered AMBA bus, accessed by the ARM through a hardware bridge and by the DSP through a DMA. The multi-tile version is based on eight Dioprisis940 tiles, with the inter-tile communication capability offered by the network processor of the system (the DNP or Decentralized Network Processor) connected to the AMBA bus of each tile and interconnecting the tiles through a NoC, a 3D Torus or a Collective Tree Network. The DNP provides a low-level RDMA API.

The flow allowed us to port a complex scientific application (the LQCD [19], Lattice Quantum Chromodynamics) as a process network. The scalability and parameters of this application give a lot of possible mappings with a significant number of required communication channels.

B. Communication paths and mapping

We are dealing with two communication channels for our examples. One is between two tasks mapped on the same tile but on the two different CPUs (RISC and DSP). The second one is between two tasks mapped on two different tiles, and on different CPUs (RISC on tile 0 and DSP on tile 1). The later implies to use the DNP for inter-tile communication.

The high level models, application, architecture including communication paths and mapping are XML-based. For instance, these are the two read and write paths for the two detailed communication paths.

\[ WrPath_{1}(ARM, RAM) = \{t0.RDM \cdot t0.ARMbus \cdot t0.BRIDGEARM \cdot t0.AHBbus0 \cdot t0.BRIDGE \cdot t0.ExtMembus \cdot t0.RAM\} \]
\[ RdPath_{2}(DSP, RAM) = \{t0.DSPmem \cdot t0.DSPbus \cdot t0.DMAdsp \cdot t0.AHBbus0 \cdot t0.BRIDGE \cdot t0.ExtMembus \cdot t0.RAM\} \]
\[ WrPath_{3}(ARM, TORUS) = \{t0.RDM \cdot t0.ARMbus \cdot t0.BRIDGEARM \cdot t0.AHBbus0 \cdot t0.DNP \cdot t0.TORUS\} \]
\[ RdPath_{4}(DSP, TORUS) = \{t1.DSPmem \cdot t1.DSPbus \cdot t1.DMAdsp \cdot t1.AHBbus0 \cdot t1.DNP \cdot t1.TORUS\} \]

The XML description corresponding to \( WrPath_{1}(ARM, RAM) \) is described in the following listing and is inserted in the architecture model.

```xml
<writepath name="t0.ARM-ARM">
  <processor name="t0.ARM"/>
  <localmem name="t0.ARMbus"/>
  <net_itf name="t0.ARMbus"/>
  <network name="t0.BRIDGEARM"/>
  <net_itf name="t0.ARMbus"/>
  <network name="t0.ARMbus"/>
  <net_itf name="t0.ExtMembus"/>
  <medium name="t0.RAM"/>
</writepath>
```

C. Communication primitives: use and initialization

Application tasks are implemented as threads of the OS. Task behaviour is coded by the user and is associated to the application model. It uses a message-passing library to communicate with the other threads of the application through channelRead and channelWrite functions as described in the following piece of code. The implementation of communications is thus transparent for the application programmer.

```c
void th1_behavior(Channel ch1, Channel ch2) {
    word b[10];
    word c = calloc(10, sizeof(word));
    ... // Send 10 words of a[]
    channelWrite(ch1, a, 10);
    // Send 10 words of b[]
    channelWrite(ch1, b, 10);
    ... // Read of 10 words
    channelRead(ch2, c, 10);
    ...
}
```

The above example, a write channel and a read channel are introduced. The write method transfers data from the global variable \( a \) in the data section and the local variable \( b \) located in the stack, whereas channel \( ch2 \) transfers data to a dynamically allocated array in the heap. Those three kinds of elements are not necessarily in the same local memory but we impose this. The communication paths are then independant from the user code. This assumption is not too restritive, as in practice, all memory sections for a given processor are mapped in the same memory.

The global variables should not be used for thread exchanges as they could be mapped on different processors.
ChannelRead and ChannelWrite are the only allowed communication primitives.

The initialization file is generated thanks to information contained in the input models. The names of the virtual files used for communications have been automatically generated (/dev/d940_rdv.0 and /dev/d940_rdma.0).

```c
int main() {
    // local variables
    Channel ch[2];
    pthread_t th1, th2;
    . . .
    // Communication channels Initialization
    ch[0] = channelInit("/dev/d940_rdv.0");
    ch[1] = channelInit("/dev/d940_rdma.0");
    . . .
    // Thread initialization
    pthread_create(&th1, NULL, th1_behavior, ch);
    pthread_create(&th2, NULL, th2_behavior, ch);
    . . .
    // End of execution
    pthread_join(th2, NULL);
    . . .
    return 0;
}
```

D. Multi-device driver synthesis

1) Driver library: Driver templates have been written in a generic way taking into account the differences between the RISC and the VLIW DSP (endianess, wordwidth, ...) and specific characteristics (and protocol) that can have an influence on the application behavior. For example, the Rendez-vous mechanism imposes a synchronization between reader and writer processes which increases the risk of deadlocks [20]. Drivers are OS-dependant, which means that all the services needed are provided by a call to an existing services of the OS. In the same way, all the accesses to hardware resources are done by calling a function provided by the HAL API.

For this architecture, we have developed four generic drivers: two for intra-tile communication (Rendez-vous and FIFO protocols) and two for inter-tile communication (RDMA [21] over the DNP and ETHERNET protocols). All of them have been written for ARM and DSP, and rely on the HAL API managing or not the DMA. Rendez-vous and FIFO manage the same devices and are thus candidates for the same communication paths. RDMA and ETHERNET do not drive the same devices. Both are multi-protocols, Eager and Rendez-vous for RDMA, TCP, UDP and Raw for ETHERNET.

2) Driver selection: An other important point not mentioned yet is performance that is function of the transported payload (the Eager protocol has better performance for small payloads whereas the Read-based Rendez-vous has more performance for large payloads, for example). These are choices that the system designer is seldom aware of, but that have to be set during the driver selection phase. The choice we made was to impose the synchronized mode for all communications (the most performing according to our benchmarks) and to alert him of the blocking nature of read and write functions.

For intra-tile, we chose the Rendez-vous driver in all cases. For inter-tile, we chose RDMA for the paths using the DNP, and ETHERNET for the paths using the MAC interface. The RDMA implements two protocols, and the choice is done at execution time depending on the payload. For the ETHERNET driver, the choice between TCP, UDP and Raw was set by hand for all channels.

3) Driver specialization: The selected driver requires now to settle the physical resources. For the following example, the virtual file /dev/d940_fifo.2 opened in the initialization file uses a shared memory whose base address is 0x301030, size of 48 words, with hardware lock number 2 and transmits integer data (information required by the DSP).

```c
uint32_t CHANNEL_D940_FIFO_DEVICES[4] = {
    /* Channel configuration /dev/d940_fifo.2 */
    {0x301030, 0x30, 2, 0 },
    /* . . . */
}
```

This supposes to pay attention to all values for each tile and memory, as well as other resources. Most of the requested values can be automatically computed, and a tool helps the system designer by providing this low level information. The tool computes values from platform settings, like memory size, number of locks, size of the allocated memory slot or some user defined values. These data are for the moment hard-coded information, but could be derived from the high-level model. Such a tool facilitates also the consistency between a read and write drivers (same virtual file name, same base address for FIFO, ...).

E. Results and future work

Several mappings of the LQCD application on one, two and eight tiles have been written and the binaries have been generated by our flow in few seconds. Table I sums up the number of channels (mapped on communication paths) and their type (intra-processor, intra-tile and inter-tile) according to the mapping. For the implementation only on one ARM processor, only one driver is used and it requires one specialization per channel on the host CPU. For a single-tile implementation on both processors of the same tile, two different drivers need to be specialized. Intra-CPU channels still require only one specialization, whereas inter-CPU channels require two different specializations, one for each processor. On eight tiles, three drivers may be used with 174 specializations (1 for each intra-cpu, 2 for inter-cpu and inter-tile). This application highlights the need of the automation concerning the software communication synthesis.

<table>
<thead>
<tr>
<th>Mapping</th>
<th>Intra-cpu channels</th>
<th>Inter-cpu channels</th>
<th>Inter-tile channels</th>
<th>Driver num.</th>
<th>Driver spec.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tile, all ARM</td>
<td>96</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>96</td>
</tr>
<tr>
<td>Tile, ARM+DSP</td>
<td>80</td>
<td>16</td>
<td>0</td>
<td>2</td>
<td>112</td>
</tr>
<tr>
<td>Tiles, all ARM</td>
<td>72</td>
<td>0</td>
<td>24</td>
<td>3</td>
<td>136</td>
</tr>
<tr>
<td>Tiles, ARM+DSP</td>
<td>56</td>
<td>16</td>
<td>24</td>
<td>3</td>
<td>136</td>
</tr>
<tr>
<td>Stiles, ARM+DSP</td>
<td>34</td>
<td>0</td>
<td>62</td>
<td>2</td>
<td>156</td>
</tr>
<tr>
<td>Stiles, ARM+DSP</td>
<td>18</td>
<td>16</td>
<td>62</td>
<td>3</td>
<td>174</td>
</tr>
</tbody>
</table>

After compilation, we can provide the size of each driver for the program memory. The Rendez-vous driver is smaller (0.05 kwords for the ARM9, 0.83 kwords for the DSP) than the FIFO driver (0.63 kwords for the ARM9, 1.08 kwords for
the DSP). The size of the RDMA driver is 1.63 kwords for the ARM9, 3.67 kwords for the DSP. The ETHERNET is much bigger in reason of the heaviness of the TCP/IP stack. For the ARM9, the size is 2.66 kwords. For the DSP, it is too big to fit in the DSP local memory. The driver with only the Raw protocol is about 2.03 kwords.

The way the communications are presented to the application programmer hides most of the low-level details of the communication infrastructure successfully. However, the adequation of the high-level model with the reality is not entirely satisfactory. Indeed, two principal factors have induced a gap between the communication path attached to a channel, and its implementation as a driver.

A first factor comes from the allocation of the used memories, which are dependant from the back-end tools (through a linker script). A second factor concerns the Net and NI components of the communication path, which may not be respected in reason of the use of a HAL by the drivers. Indeed, the HAL imposes its own way to reach the close environment of the CPU (memories, NI, ...), and the way the communication path defines the blocks which are traversed during a read or write operation may thus not be taken into account. One way to respect this specification would be to generate a part of the driver according to the communication path. This kind of generation requires a more detailed representation of the architecture and the peripherals put into play, and could be an interesting continuation of this work [22].

A future work consists in exploiting the adopted formalism (in particular the hardware communication paths) through the introduction of an intermediary formal model close to the one defined in [23] which could verify the respect of the communication paths and analyze the performances of a given mapping according to specific constraints like memory access cost or network capacity.

VII. CONCLUSION

We have presented in this paper a formalism focusing in the representation of hierarchical platforms communication infrastructure. In particular, we have introduced the concept of hardware communication path to represent a possible communication channel between a local memory and a communication media for a message-passing paradigm. We have associated this representation to a simple application model and a mapping of application functionalities into the architecture elements. We have detailed our communication synthesis approach, based on a driver library. Our drivers are multi-device and generic, their selection is based on the matching of the driver’s device list with the hardware communication path description chosen for an application channel in the mapping file. We described the specialization of each driver instance, which allowed us to obtain a functional and semi-automatic binary generation flow.

This generation flow was successfully submitted to a scientific application programmer who ran several mappings of his application in a painless way.

ACKNOWLEDGMENT

This work has been mainly funded by the European Community through the Sixth Framework Program (FP6 SHAPES project under the Grant Agreement number 26825) and the Seventh Framework Program (FP7 EURETILE project under the Grant Agreement number 247846) when Alexandre Chagoya-Garzon were part of the staff of TIMA lab.

REFERENCES