Self-reconfigurable embedded systems: from modeling to implementation

Guy Gogniat, Jorgiano Vidal, Linfeng Ye, Jérémie Crenne, Sébastien Guillet,
Florent de Lamotte, Jean-Philippe Diguet, Pierre Bomel
Université de Bretagne Sud - UEB, Lab-STICC - CNRS, UMR 3192
Centre de Recherche - BP 92116, F-56321 Lorient Cedex - FRANCE

***Abstract*—Self-reconfigurability is becoming a reality from an hardware point of view. Many studies have shown the benefit of such a technology which allows greater flexibility, performances and cost reductions. However there are several points that still represent major challenges: i) design flow and associated tools as current solutions are too tightly connected to hardware platforms. A need for abstraction is strongly required to allow designers to build and evaluate efficient systems. ii) architecture models and associated portfolio of reconfigurable IPs as most of current solutions are based on ad hoc approaches which lack of reusability and portability. A more systematic design methodology associated to an efficient architecture model is required for a large adoption of such a technology. iii) bitstreams repository since most available solutions are based on a bitstreams library stored in Flash memory. Such an approach is not meeting scalability requirements and a more global solution is mandatory with a clear hierarchy of bitstreams repository. In this paper a comprehensive methodology is presented in order to address these three major points. Our solution leads to an efficient approach from modeling to implementation of self-reconfigurable embedded systems.**

I. Introduction

Advances in reconfigurable technologies allow entire multiprocessor systems to be implemented in a single FPGA (Multiprocessor System on Programmable Chip, MPSoPC). Designing such systems with existing tools will soon become unmanageable due to complexity and productivity reasons. One promising solution to mitigate designer task is to further increase abstraction levels in order to hide many implementation details. Such an approach will allow system designer to have access to various SW and HW technologies without being an expert of all of them. Several efforts have been performed these last years to promote UML (Unified Modeling Language) [1] in order to consider it as an efficient language to model multiprocessor systems. Considering a single language to design these systems is very interesting and allows to speed up design time and system integration.

To build an efficient design flow it is mandatory to rely on a well defined model of platform. Typical MPSoPC systems (Figure 1) are composed of a set of embedded processors executing tasks, which communicate together to implement the system functionality. Specific IPs (Intellectual Property) can be considered to speed up the computation of intensive tasks. These IPs can be used as co-processors directly connected to processors or accelerators connected to internal buses. During the development process, flexibility, performance and area correspond to key concerns for the designer. Existing FPGA technology (e.g. Xilinx Virtex devices) offers dynamic and partial reconfiguration (DPR) features that can be advantageously considered to address these points. DPR allows HW tasks to share the same resource if their execution is exclusive. Such a solution is very interesting as it reduces the total system area while still meeting performance constraints. Supporting DPR allows multiprocessor systems to replace co-processors or accelerators at run-time, it can significantly increases platform flexibility. Unfortunately there is a lack of tools addressing the design of reconfigurable MPSoPCs at the abstraction level mentioned above. Thus in this paper we propose to reduce the gap between the design of reconfigurable MPSoPCs and the targeted technology (FPGA).

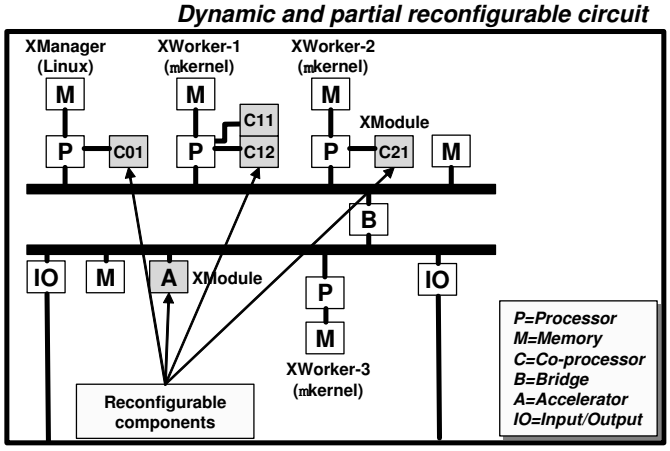

Fig. 1. Reconfigurable multiprocessor architecture model

The rest of the paper is organized as follows: In Section II we discuss existing efforts in the domains of reconfigurable systems/technology and also in the domain of embedded systems modeling. In Section III, the reconfigurable multiprocessor architecture model is described. In this section, both hardware and software components are detailed. In Section IV the bitstreams repository hierarchy is presented. This section illustrates how providing an efficient infrastructure to deliver bitstreams to the system. Section V focusses on the modeling of embedded systems and on code generation. This part shows how guiding designer to provide more rapidly and with more reliability a system. Section VI illustrates our approach through an example. Finally Section VII concludes the paper.

II. Related work

A. Reconfigurable systems

Numerous experiences have been carried out in the domain of reconfigurable architectures. The RAMSoC [2] is an interesting project, where are addressed the two main drawbacks of traditional approaches. The first one is the necessity to find a trade-off between homogeneous and application-specific MPSoC. The second one is a meet-in the middle design methodology to offer run-time configuration capabilities. This work is related to an architecture model with various processor types. The proposed solution is based on soft-processor (Microblaze) with configurable accelerators that can communicate through a configurable network on chip. The MOLEN [3] reconfigurable processor uses microcode and custom configured hardware to improve performance, which allows the programmer to modify the processor functionality and hardware without architectural and design modifications. However the focus is not the design flow but the platform. In both previous projects, there are no reference to a programming model with standard API, which are nevertheless required to transparently use processor, HW accelerators or co-processors.

The hArtes Approach [4] addresses the development of an holistic tool-chain for reconfigurable heterogeneous platforms. The entire tool-chain consists of three phases: Algorithm Exploration and Translation, Design Space Exploration and System Synthesis. The objective of the hArtes design flow is to automate the rapid design of heterogeneous embedded systems. But from the reconfigurable application designer's point of view, it's rarely necessary to design a specific reconfigurable MPSoC platform, that is why we propose a rapid application development based on multi-processor reconfigurable systems.

B. Reconfiguration technologies

There have been several works dealing with reconfiguration through a network. Lagger et al. [5] propose the ROPES (Reconfigurable Object for Pervasive Systems) system, dedicated to the acceleration of cryptographic functions. It is built with a Virtex2 1000 running at 27 MHz. The processor is a synthesized Microblaze executing uCLinux's code. It downloads bitstreams via Ethernet with HTTP and FTP protocols on top of a TCP/IP stack. For bitstreams of an average size of 70 KB, DPR latencies are about 2380 ms with HTTP and about 1200 ms with FTP. The reconfiguration speed is about 30 to 60 KB/s, be a maximum of 17 Kb/(s.MHz). Williams and Bergmann [6] propose uCLinux as a universal DPR platform. They have developed a device driver on top of the ICAP. This driver enables to download the content of bitstreams coming from any location because of the full separation between the ICAP access and the file system. Connection between a remote file system and the ICAP is done at the user level by a shell command or a user program. When a remote file system is mounted via NFS/UDP/IP/Ethernet the bitstreams located there can naturally be downloaded into the ICAP. The system is built with a Virtex2 and the processor executing the OS is a Microblaze. The authors agree that this ease of use has a cost in terms of performances and they accept it. No measures are provided. To have an estimation of such performances, some measures in a similar context have been done. A transfer speed ranging from 200 KB/s to 400 KB/s has been measured, representing a maximum performance of about 32 Kb/(s.MHz).

C. Modeling approaches

The use of model based approaches for codesign has been discussed in [7], which pointed out some advantages: cost decrease, silicon complexity handling, productivity increase, etc. Several works have also shown the benefit of using UML for embedded system modeling [8], [9], [10], [11], [12]. However most of them define specific profiles to model embedded systems. Using specific profiles limits a large adoption of these approaches as they do not rely on a standard. Furthermore proposed approaches mainly target system analysis and simulation. Dynamic and partial reconfiguration modeling using UML allows such methodologies to take advantage of dynamic reconfiguration capabilities of moderns FPGAs. Although there is a lot of work on embedded system modeling using UML, only few explore dynamic and partial reconfiguration capabilities [13], [14].

In [13], authors use UML sequence diagram with specific stereotypes to model dynamic reconfiguration. Their approach is very simple and efficient, but it lacks platform modeling. In their work the system platform is fixed: a processor with a reconfigurable device as an auxiliary computing unit. Also, it does not support dynamic and partial reconfiguration.

In [14], authors detail a dynamic reconfigurable system by extending UML/MARTE with specific stereotypes. Their approach is developed in a design environment called GASPARD, where VHDL code is generated. This approach is very target-dependent and requires a strong level of expertise as all elements of the Xilinx partial reconfiguration design process need to be modeled.

D. Contribution

Compared to existing efforts, our contribution takes place at three levels: i) architecture model from an hardware and software point of view, ii) bitstreams repository hierarchy and iii) modeling and code generation for reconfigurable systems. Our approach relies on communication between a master processor running Linux and specialized slaves, without OS, executing computation intensive tasks. Our objective is to decide configuration online and to provide synchronization solutions between master and slaves and to execute programs where registered functions are called. So our contribution, regarding this aspect, is independent from the OS choice and mainly related to the way a master can fire and specialize slaves with specific computing intensive tasks. Concerning modeling of reconfigurable systems, our approach only uses standard UML/MARTE elements. We have defined specific properties, which are required by the code generation tool that is target dependent. However, in order to allow a large adoption of dynamic and partial reconfiguration we hide

from system designer many technological details. Furthermore, reconfiguration services and resources are automatically added to the system during the code generation step and are based on predefined APIs.

III. Architecture Model

In order to be able to provide an efficient and reliable solution for designers it is fundamental to define a generic architectural model which can be tuned depending on the application requirements and associated constraints. In our case we propose a multiprocessor reconfigurable architecture with several computation nodes connected through shared memories.

A. Reconfigurable multiprocessor architecture model: XPSoC

The XPSoC is based on a master processor (XManager) and several slave processors (XWorker) and/or several reconfigurable IPs (XModules). A XModule can be either a standalone HW accelerator connected to the internal system bus or an HW coprocessor directly connected to a slave processor (XWorker). This model has been designed for data-flow applications where intensive computations are required. Communications between processors are implemented through shared memories (for data) and message passing (for control). The XManager runs Petalinux and deals with applications execution management (e.g. loading new configuration, starting new task, synchronization between tasks). Each XWorker can execute a single task at a time, this task is provided by the XManager depending on execution requirements. When a XWorker is in an idle mode, it pends on its shared memory waiting for some new tasks to be executed.

A general definition of the XPSoC is defined by a reference model $MxSy_iSy_{i+1}Sy_n$, where Mx is a XManager with x places for reconfigurable coprocessors, S means a XWorker with y_i coprocessor places. In case S is not a XWorker but an hardware accelerator (XModule), no reconfigurable places are mentioned.

Figure 1 shows an example of a given architecture model composed of one XManager and three XWorkers. The three XWorkers have different reconfigurable coprocessors. These coprocessors, which are acceded through FSL links when the softcores are Microblazes, are configured by the XManager at run-time. Note that Xilinx partial reconfiguration flow enables execution/reconfiguration overlapping, thus the XManager can reconfigure an unused coprocessor without freezing the XWorker. The objective of the reconfiguration decision is to optimize execution time according to applications needs.

B. Software architecture and synchronization

An application is modeled as a set of tasks which can be executed in different ways. The first one is a pure SW execution by the XManager as a Linux thread. The second solution is a SW execution of the thread on a XWorker. The third method consists in running the thread on a XWorker with a standard function implemented on coprocessor. The last possible implementation is when the whole thread can

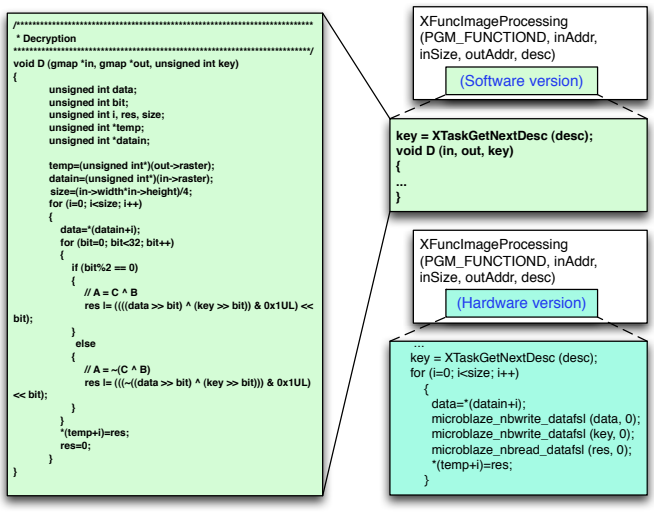

Fig. 2. XFunc prototype example for Hardware and Software tasks

be executed by a dedicated hardware accelerator. For each version, a software binary file and/or a bitstream are required. Considering that most connected embedded systems are based on standard functions, we assume they can be available in the system memory or from remote configuration servers as will be described in Section IV.

For each XWorker, a configuration table is defined in a memory space shared with the XManager. This table contains records used for XWorker/XManager synchronization. The first record provides global parameters such as architecture model ID (XM_{ID}), XWorker status and input and output addresses and sizes. The second record is the queue of tasks to be executed. Each task is specified with addresses pointing on HW and SW versions of task binaries, with input and output data buffer addresses and sizes. Then a record is added for each coprocessor, it mainly contains the standard function ID (FU_{ID}) that also indicates I/O data formats.

This hardware model is efficient but is not sufficient from a software designer point of view. Indeed, there is still a need, at the application level, for programming models and communications APIs. These APIs must enable designers to easily map applications over many different possible reconfigurable architectures without tedious rewriting, while at the same time ensuring efficient code production. To cope with this issue and simplify XPSoC application development and reuse of reconfigurable IPs (XModule), we propose two API libraries: XTask and XFunc.

C. APIs for reconfigurable application

Considering productivity constraints and debugging overhead, the software engineer cannot and must not spend too much time to understand details about configurable coprocessors and accelerators or on-the-fly partial reconfiguration steps. So at the application level, there is a need for programming models and communications APIs that make a clear separation of concepts. This is the objective that motivated the development of XFunc/XTask API libraries.

1) XFunc API: XFunc is a set of generic functions that can be specialized to handle various application classes (Video, Audio, Network, etc). XFunc provides a unified programming prototype for hardware version and software version. XFunc is specified as XFunc_Class_of_Applications(parameters). The objective of the XFunc API is:

- Offering application designers a solution to develop applications without a strong understanding of the complexity of DSP algorithms or underlying hardware.
- Changing critical function (e.g. SW or HW Audio / Video codecs) within the involved class of application without modifying the code at application level.
- Adapting any application code to a reconfigurable MP-SoC system.

Various implementations of a given XFunc, corresponding to various performance/area trade-offs, may be available. However the API for function calls must remain unchanged for a given application domain. Figure 2 illustrates a simple example of an XFunc API for decoding encrypted PGM images. XFuncImageProcessing is a generic API for image processing to be executed in hardware or software with the following parameters. PGM_FU_{ID} is the function ID of PGM image processing in this example, $inSize$ is the size of the input buffer which has the following memory mapping $inAddr$, and $outAddr$ which corresponds to the memory address of the output buffer, finally $desc$ is a pointer to a specific structure used in this case to get the key used for image encryption.

2) XTask API: XTask is an API that supports reconfigurable application programming in C on XPSoC. It consists of a set of library routines and environment variables that create and manage tasks to be executed on XWorkers. The main services offered by XTask API are:

- Creation, suspension, destruction of an XTask.
- Synchronization between XManager/XWorker.
- Management of XModules (spots of reconfigurable resources).
- On-the-fly reconfiguration decision.
- Management of dynamic partial reconfiguration.

The XManager creates a specified number of XWorker tasks (XTask). An application can be composed of none, one or multiple XTasks. An XTask runs concurrently, with the runtime environment allocating hardware accelerators to different XModules. In order to manage the reconfigurable resources and the reconfigurable XTasks, we define a set of functions. The following is a brief description of XTask main API:

- *XTaskCreate* creates a XTask for FU_{ID} function in main memory
- *XTaskRunBlocking* copies XTask to shared memory and launches a blocking task.
- *XTaskUpdateFuntionList* updates the list of function to be implemented in hardware by order of priority.
- *XTaskUpdateSystem* updates the XPSoC architecture by performing dynamic partial reconfiguration if necessary.

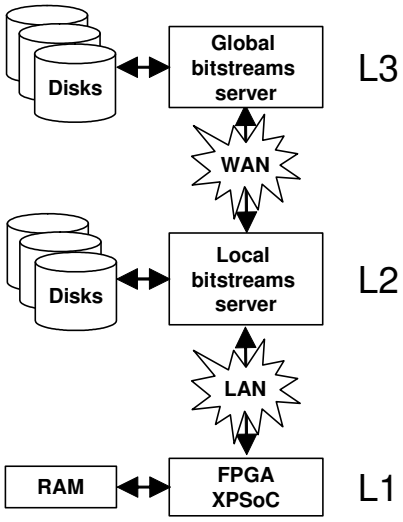

Fig. 3. Bitstreams repository hierarchy

IV. Bitstreams repository hierarchy

XPSoC relies on the availability of bitstreams at runtime in order to dynamically adapt the system. Thus, a bitstreams repository becomes necessary in order to manage self-adaptivity. Furthermore, bitstreams repository can contain a large number of configurations and in some cases reconfiguration time can be critical, thus a hierarchy of repositories is required. The repositories must communicate through adapted physical channels and network protocols with the partially reconfigurable FPGAs. The bitstreams repository hierarchy has to deliver all IPs to targeted XPSoC implemented on FPGAs. In a typical network topology, this hierarchy is composed of three levels (Figure 3):

- L1: a board local bitstreams cache in memory.
- L2: a fast bitstreams server located in a dedicated LAN using a data link level protocol.
- L3: a standard global slower server located anywhere and accessed via TCP (Transmission Control Protocol) or UDP (User Datagram Protocol) based protocols.

A. Hierarchy Level L1

Level L1 is the board level where designers glue together FPGAs and Flash or RAM memories. Bitstreams can be stored in memories and it is very common to use 512 MB Flash ones. This is the most popular way to store bitstreams and build prototypes. But the less memories there are, the cheaper the system is to produce in high volume. L1 is geographically the closest repository to the FPGA, and the one with the smallest latency (in the range of ms).

The PR (Partial Reconfiguration) community agrees on the fact that, in applicative domains with strong real-time constraints, PR latency is one of the most critical aspects in its implementation. If not fast enough, the PR interest to build efficient systems can be jeopardized.

Depending on the system designer's ability to build an efficient data pipeline from the bitstream storage (RAM, Flash,

or remote) to the ICAP, the performances will be close (or not) to the peak values. Maximum downloading speed rate announced by the fabless in internal reconfiguration mode is capped to a maximum of 800 Mb/s when ICAP accesses are 8 bits wide.

1) Cache Architecture: The use of a Flash memory via a mass storage card or an integrated on board memory is well known and used, at least for boot time. This kind of non-volatile storage is useful for maintaining a large range of bitstreams, and when the access time is not a constraint. Without talking about writing transactions, reading is close to 500 cycles for a single 32 bits word. This value is of course dependant on the Flash technology, and the associated controller. With the use of a cache, designers are able to solve this issue by copying a bitstream into a faster memory which is located closer to the XManager in our case. The problem here is that partial bitstreams are in a range of hundredths of KB. Then, BRAMs memories are not the answer due to the overall available blocks which are much lower. BRAMs are a very scarce resources in FPGA. In our approach, we propose the use of an SRAM for a cache memory. It is a tradeoff between the faster volatile memory (BRAM) and the lower non-volatile memory (Flash). This solution is efficient to speed up reconfiguration for some critical bitstreams at a low memory cost. The policy about memory usage is LRU (Least Recently Used) based, which means the less used bitstreams will be replaced by the most used ones if there is not enough memory space to store all bitstreams.

B. Hierarchy Level L2

Level L2 is the LAN level with a specific data link level protocol. It can provide a reconfiguration service with an average latency of 10 ms. Ethernet, in its simplest usage, is a medium sharing mechanism on top of which many protocols have been added. But it can also be seen as an excellent serial line. In terms of buying cost and ease of deployment it is a prime candidate to transfer bitstreams between close devices like our FPGAs and the LAN bitstreams server.

When looking at the state of the art, it appears that "Microblaze + Linux + TCP" dominates. Unfortunately, best downloading speeds are far below the ICAP and network maximum bandwidth.

1) Data Link Over Ethernet 100 Mb/s: Ethernet standard IEEE 802.3 is now a rich set of communication technologies to build cost effective LANs and to connect computers together. It is based on the diffusion of packets on a shared medium with collision detection (CSMA-CD). The insertion of switches and hubs (multi-ports repeaters) to simplify cabling and to improve speed and quality of services, transforms the LAN into a set of point to point links connected through LAN-level routing equipments. With this topology, two equipments connected to the same switch communicate through a quasi-private link (excepted for the broadcast packets). Ethernet's evolution is such that tenths, and even hundredths, of Mb/s are now available at very low costs and with quasi-null error rates.

2) Ethernet access: With our repository hierarchy the bitstreams server is connected to the same LAN than our platform, it does not need level 3 routing toward any other LAN. Therefore we do not need IP routing and its companion protocols such as ICMP, ARP, TCP and UDP. The immediate drawback is that it does not allow the downloading of a bitstream from any other machine over Internet, this is the function of the L3.

C. Hierarchy Level L3

An ad-hoc specific data link is useful when no IP routing is required and only a little amount of hardware and software resources is available. However it is also necessary to be able to download a bitstream from any machine. The use of the standard network architecture TCP/IP fits perfectly when a remote reconfiguration is necessary. Level L3 is the WAN (Wide Area Network) level where the latency is about 100 ms because of its geographic position which is the farthest.

1) Common Used Transport Protocols: Rind et al. [15] describe choices for TCP (Transport Communication Protocol) over UDP (User Datagram Protocol) and vice-versa related to speed, numbers of mobile devices and link capacity (bandwidth) metrics. Results are given in terms of throughput and goodput via a network simulator. It shows that TCP is giving better performance when minimum number of mobile devices are connected to a WLAN (Wireless LAN) and clearly setup that faster moving nodes are highly disturbing packets transmissions. UDP is found better if it is possible to bear little loss of packets. Consequently it is a first choice protocol for fast delivery of data. As the system we target will be using a WIFI link and thus is limited to a much lower throughput, very high Gigabit transfer rate is oversized. Ploplys et al. [16] perform a study where "wireless" UDP is used for real-time performance in control. Loss of data is well defined, explained and evaluated based on many factors such as range, environmental obstacles, computational loads and increased network traffic. Existing work establishes that TCP is vastly employed in LAN topology and UDP in WLAN. The use of UDP is thus natural when targeting wireless handled devices. UDP is also the most suitable standard for systems with a high latency and needs by nature, a shorter communication time.

2) TCP/IP Architecture Model: For a wireless environment where bit error rate is high, TCP performances are highly degraded due to its window based congestion control mechanism. UDP is similar to TCP and stands in the same TCP/IP layer. Known UDP applications are DNS and SNMP. Connectionless, its difference is located in the relationship between two parties. In other words, one can send data to another and that is all. UDP doesn't provide any reception reliability thus, there is no guarantee that packets will arrive. However if the transmission is correct, the packet will be received without any data corruption. UDP is faster than TCP as there is no extra overhead for error-checking above the packet level. A comparison between TCP and UDP is given in Table I. In the context of bitstreams downloading from a wireless server, UDP provides better performances over TCP

and in case some errors occur during transfers, the whole bitstream is resent. This solution is acceptable as bitstreams size is low compared to network capacity.

D. Hardware Architecture

In our approach XManager (a Power PC PPC405 or a Microblaze) executes the reconfiguration service. The Ethernet PHY controller is connected to the PLB and uses an integrated DMA (Direct Memory Access) to speed up transfer of incoming packets to the cache memory. A second DMA can be instantiated and managed by the Xmanager itself in order to copy a bitstream in cache to the ICAP removing PPC405 software copy bottlenecks. Finally the ICAP, connected to the OPB, manages the access and the downloading of bitstreams into the reconfigurable areas.

In this study, the XManager (PPC405) instruction cache was activated and set to 16 KB large (8 BRAMs). When enabled the data cache is also 16 KB large. Table II demonstrates that software data copy with data cache enable is the best setup. This can be explained because EDK's DMA engine has no internal buffering, and doesn't perform burst transfers. For processor without instruction cache, it might make sense to add a DMA, otherwise the inner loop of the optimized memory copy would be in cache and be executed at 2 cycles per instruction. The limiting factor will become the OPB latency (reading/writing from/to the OPB RAM). Indeed, when a data cache is enabled and as the processor exhibits cache coherency anomalies, it has to remain clean. It is the responsibility of the developer to ensure that any buffers in cache which are passed to the DMA are flushed from it.

E. Software Architecture

Onto the client FPGA, the XManager runs an executable built with a PPC GNU GCC and a TCP/IP stack.

1) lwIP as a TCP/IP networking stack: Instead of developing a networking library from scratch, we choose an open source TCP/IP stack designed for embedded systems: lwIP. Directly available in EDK, lwIP [18] is an implementation under BSD licence of the TCP/IP stack with RAM usage friendly in mind. The porting proposed by Xilinx in EDK is quite robust (both Microblaze and PPC can be used without any problems). lwIP is also featuring a quite exhaustive characteristics list (IP, ICMP, ARP, UDP and TCP) and can be run with an underlying OS or not.

Our first approach was to tailor lwIP to use UDP only as we don't need another protocol. To ensure packets producer-consumer paradigm, lwIP stack uses a pool of buffers. This pool is a critical point in terms of performances and goodput and has to be well scaled. Default setting value for this segment is close to 800 KB large be 512 packets of 1528 bytes. With that consideration we found native lwIP parameters to be oversized in EDK. The absence of transmission errors during days of testing, which consists of sending and receiving data as fast as possible and checking right packet order, proves that reducing it to 100 KB is pertinent without interfering overall performances.

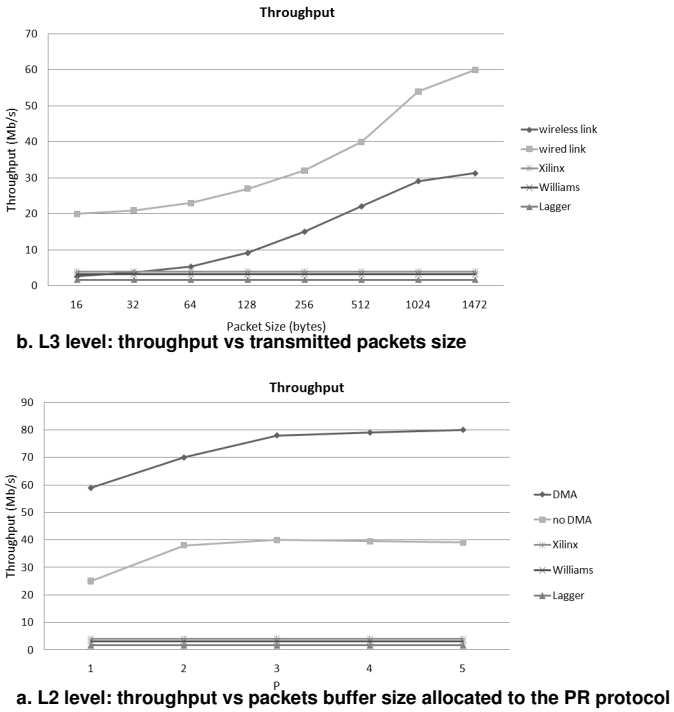

Fig. 4. Performances of bitstreams repository hierarchy servers for a XManager running at 100MHz

2) Software DPR Protocol: The protocol is able to work in two modes: slave and master. In master mode, the FPGA is responsible for asking the server a partial bitstream. In slave mode, it reacts to the server requests and is forced to update itself. Obviously, obtaining a maximal reconfiguration throughput must be considered with care. Safety concerning the write of a partial bitstream to the reconfigurable area is necessary in partial reconfiguration context. A loss of a packet will result in an incomplete form of data reception, so on an impossibility of writing the complete partial bitstream into the reconfigurable area. Manifestly, it will lead to an unpredictable behavior. To avoid this, a frame number helps to know if a packet is missing or a wrong received order reception occurs. In addition, before writing to ICAP, a CRC (Cyclic Redundancy Check) is done to be sure that everything was fine during the transmission and every packets was received correctly without transmission errors.

F. Performances

At the L1 level, with our solution we can reach an average download of bitstreams from the cache to the ICAP of about 2.1 Mb/(s.Mhz) be 210 Mb/s when the XManager (PPC405) is clocked at 100 MHz. The cache is configured to store 16 cachelines of 32 slots of 1496 bytes. With partial bitstreams of 74 KB, it is possible to store 10 bitstreams. The throughput decreases when the number of requested bitstreams is higher than the cache capacity. When no required bitstream is found in the local cache the hierarchy level L2 is automatically queried. This level is able to give access to a large number of partial bitstreams using a local server. At the L2 level, results

Protocol	Complexity	Speed	Architecture	Caveats
UDP	Low	High	Broadcast Client/Server	Unreliable, String data
TCP	Average	Low	Client/Server	String data

TABLE I
COMPARISON BETWEEN TCP AND UDP PROTOCOLS [17]

	Cache Enable	Cache Disable	Cache Disable + ICAP DMA	Cache Enable + Both DMAs
throughput	-	+	-	+
hardware memory footprint	+	+	-	-
software memory footprint	+	+	-	-
overall	++	+++	-	+

TABLE II
HARDWARE/SOFTWARE PARTITIONING OPTIONS RESULTS

obtained (Figure 4.a) depend on packets buffer size allocated to the PR protocol. The curves at the top represent measured speeds for our solutions (with or without DMA). Maximum speeds of 400 Kb/(s.MHz) and 800 Kb/(s.MHz) can be reached when the adapted packets buffer size is set up (9 KB, 6 x $1.5KB$). Compared to usual buffer pools of hundredths of KB for standard protocol stacks, this is a very small amount of memory to provide a continuous PR service. Flat lines curves at the bottom, represent the average speeds reached by Xilinx, Lagger and probably Williams. Our PR protocol exhibits a reconfiguration speed of 80 Mb/s closer to our local 100 Mb/s Ethernet LAN limit. At the L3 level Figure 4.b sums up throughput results. Results obtained depend as we could expect, on transmitted packets size. We obtain a sustainable 60 Mb/s throughput with an average packet size of 1492 bytes. This high transfer throughput matches with WIFI WLAN rate where "only" 30 Mb/s is reachable.

V. MODELING OF MULTIPROCESSOR RECONFIGURABLE SYSTEMS

The next step in our approach for building reconfigurable systems is to define a design methodology that can capture both multiprocessor architecture model and bitstream downloading services. To achieve that goal our methodology takes into account three main elements to be modeled: **the application**, **the platform** and how application fits into the platform, what we call **the allocation**. As previously described, the application is defined as a set of communicating tasks where each task performs some computations. The platform is defined as a set of connected IPs (XManager, XWorker, XModule) and the allocation is a mapping between application tasks and platform components. The allocated model is used to generate the final system. Our approach only uses standard UML/MARTE elements. Embedded C code is used to implement software tasks. The code generation tool supports the set of transformation rules used to perform the system generation from the three previous models.

A. Modeling concepts

We achieve dynamic and partial reconfiguration through the allocated model, which explores a reconfigurable platform by performing a mapping between an application and a reconfigurable platform.

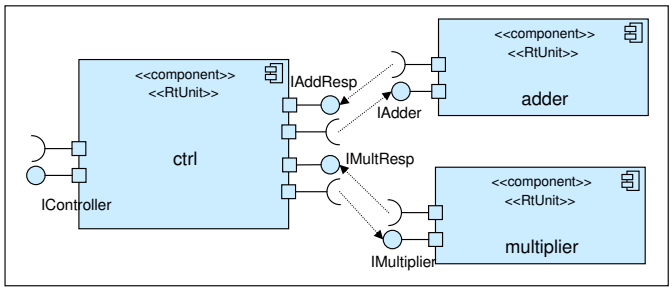

Fig. 5. Components model for application tasks

1) Application: The application is defined as a set of tasks that perform computations and communicate to exchange data. Each task (XFunc) receives data, performs some computations and sends results to other tasks. Communications among tasks are done by events. An event contains a type and associated data. Upon an incoming event, a task performs computation and sends results to other tasks.

In order to model tasks using our methodology we use UML components. Each component behavior is performed by classes instances. The **main** behavior of a component is described by an active class, whose behavior is defined by a state machine. The only way to communicate with a component is by sending/receiving events to/from the component. The events (UML signals) must be sent through ports, that offer or require a service, defined in UML by means of interfaces.

Figure 5 shows an example with three components: one for a controller task, responsible for the control flow and the scheduling, and one component for each operation. Our controller task is used to send/receive events to/from the co-processors (XModule) and to communicate with other processors (XManager or XWorker) or external components. The controller task receives the inputs and triggers both operations, one after the other. All communications are done by UML ports and use only UML signals, by sending events. Each port contains an associated interface, which will be explained later.

Each component contains the MARTE HLAM RtUnit

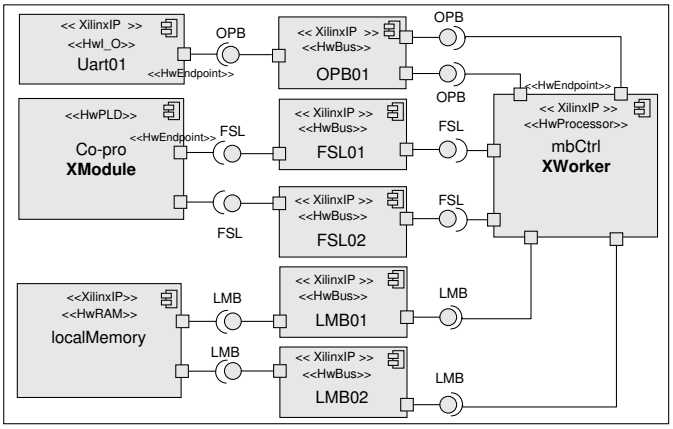

Fig. 6. Platform components modeling

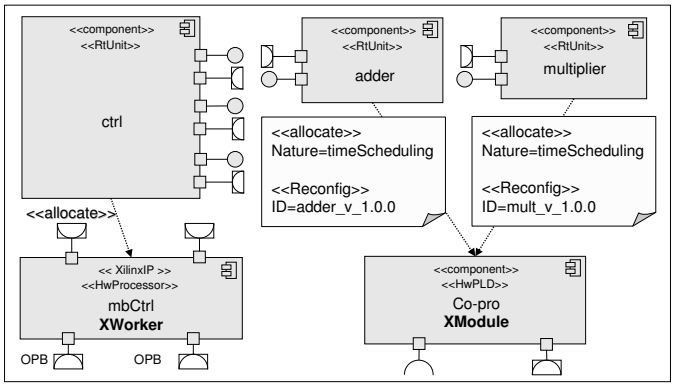

Fig. 7. Allocation of the application on the platform

stereotype. The RtUnit indicates that the component is a real-time unit. Such information allows allocation of the application components to the platform components.

2) Platform: The platform is defined as a set of IPs that can be reused from an IPs library (XManager, XWorker, XModule).

UML components are used for platform modeling, where each component is an IP. We use UML MARTE HRM profile elements to identify the IP characteristics e.g. version, name and address [19]. Thus specific stereotypes are used for each IP library. Figure 6 shows a typical platform in UML which contains one processor which can be a XWorker (`mbCtrl`), a co-processor which is a XModule (`co-pro`), an input/output component (`Uart01`), and a memory (`localMemory`). These components are connected together by buses (`OPB01`,`FSL01`,`FSL02`,`LMB01` and`LMB02`,). All the components except the co-processor contain the ≪XilinxIP≫ stereotype, with specific information for IP reuse. The co-processor can be dynamically replaced as we use FPGA dynamic and partial reconfiguration property. To identify such a property we use MARTE HRM HwPLD stereotype.

The reconfiguration service (XTasks API, XTaskUpdateSystem) is not modeled by the designer in order to abstract the implementation details of such a technology. The service instantiation is handled by the code generation tool. This approach allows the model to target different technologies.

3) Allocation: Allocation is the key step in our methodology as it models the complete system to be implemented. Once the behavior is defined in the application model and the execution structure is defined in the platform model, we need to map the behavior to the platform, what we call the allocated model. This step is performed manually. All application components stereotyped **RtUnit** (from MARTE HLAM profile) are allocated to a platform computing component (processor or PLD, as defined in MARTE HRM profile).

Figure 7 shows a part of the allocated model of our example. The controller task is allocated to the processor (XWorker in that case) and the operation tasks are both allocated to the same co-processor (XModule). The allocation of two application components to a same non programmable but reconfigurable platform component with some specific information indicates reconfigurability and is explained in the next section. Our example requires two IPs: *adder* and *multiplier* that share the same HW area. This is a design decision, as their services are not requested at same time.

4) Reconfiguration: Our approach to model reconfigurability is done in the allocation step where the application is mapped to the platform. We allow co-processors to be reconfigured at run-time. In Figure 7 two RtUnit are allocated to the same co-processor. Specific stereotypes are used to indicate the reconfiguration modeling.

An allocation is an UML dependency stereotyped with MARTE ≪allocate≫ stereotype. *Nature* tag must be set to *timeScheduling* in order to inform the code generation tool that dynamic and partial reconfiguration is required to share HW area. For each allocation an ID needs to be defined, which is used by the reconfiguration service (XTasks API, XTaskUpdateSystem) to identify the right bitstream to be loaded upon request.

The HwPLD stereotype is used to indicate a reconfigurable component in the platform. This information is used by the code generation tool to generate the right information for the synthesis tools used to implement the system on the target platform (FPGA). We have added the `Reconfig` stereotype to the allocation, to allow the code generation tool to insert the reconfiguration service call in the processor code. The `ID` tag is used to perform bitstream selection and downloading.

Figure 8 shows the generated code from a state of the controller that sends an event to a co-processor. The code generation tool searches in the controller state machine all events sent to co-processors. For each event it checks if it is sent to a reconfigurable component. If true a reconfiguration request is inserted just before sending the event. The reconfiguration command provides the bitstream `ID` to the reconfiguration service. The XTaskUpdateSystem command is offered by the reconfiguration service (XManager). Of course if the configuration is already active, the XManager does not reconfigure the component.

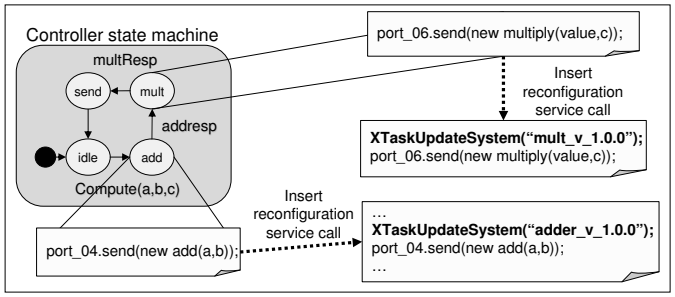

Fig. 8. Allocation with specific stereotypes models dynamic reconfiguration

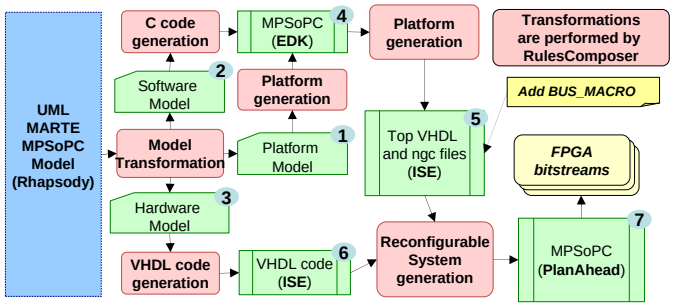

Fig. 9. Flow and used tools: The code generation tool generates projects and scripts

B. Development flow and tools

As we consider a high abstraction level for modeling we require implementation steps into existing technology and tools. Our development flow relies on existing tools for UML modeling, model transformation, embedded platform design, HW synthesis and bitstream generation. Xilinx FPGAs and associated tools are used, as they offer dynamic and partial reconfiguration technology. Figure 9 shows the development flow and considered tools.

Rhapsody UML modeler is used for UML modeling. RulesComposer, a Rhapsody plugin, is used as model transformation/code generation tool. Through RulesComposer we can define transformation rules, that take as input an UML/MARTE model and generate implementation models. We have defined implementation models for platform, VHDL and C code. Code generation facility is built from the UML model, where a set of transformation rules creates a specific target model. As the UML model contains the complete system design, the first step is to extract the various system parts: platform with existing IPs (1), software code (2) and new IPs (3).

The platform and software models are used to automatically generate a Xilinx EDK project [19] (4), which is the Xilinx tool used for embedded system design. EDK defines a platform as a set of IPs and each new IP is generated with an empty behavior. Each processor (XManager and XWorker) contains a software project associated to it. The C model is used as input to generate processor software code in EDK. The code generation tool automatically inserts in the EDK project several IPs required to perform dynamic reconfiguration (i.e. ICAP, Ethernet controller). The EDK project is automatically transformed into an ISE project (5). Xilinx ISE is the tool used for hardware design, which accepts VHDL as input language. The EDK project exports a top VHDL file with all the components of the system.

Each VHDL model (new IPs) is used to generate ISE projects (6). At that step of the design flow we have one ISE project for the system and one VHDL project for each new IP. Then in the Xilinx flow we update the VHDL files with reconfiguration information. Each reconfigurable component must be connected through `BUS_MACRO`. A `BUS_MACRO` is a placement and routing constraint used to allow partial reconfiguration area in the target Xilinx FPGA. `BUS_MACRO` is inserted in the top VHDL file by the designer[1]. The set of ISE projects is used to generate a set of files that contains logical design data and constraints: the NGC files. Each NGC file is an IP that must be placed in the FPGA.

The NGC files are used as input to generate the PlanAhead project (7) which is required to generate the set of bitstreams. With PlanAhead we place all IPs in the FPGA and mark which areas of the chip are dynamically reconfigurable. Then PlanAhead generates one bitstream for the initial configuration and a set of partial bitstreams for each possible configuration. Also, an empty bitstream is generated for each reconfigurable area in the design (for each co-processor (XModule) in our architecture).

In our example, from Figure 7, 4 bitstreams were generated: **static_full.bit** (initial system configuration), **copro_add.bit** (adder component), **copro_mult.bit** (multiplier component) and an empty bitstream, **copro_blank.bit**.

Each bitstream is stored in a bitstream server on the network and contains an `ID`. At system start up the `static_full.bit` bitstream is loaded into the FPGA (with the empty co-processor by default). The reconfiguration service present in the system contains information about how to load the bitstream and where to load it from. Reconfiguration commands (XTaks API) inserted by our code generation tool uses the reconfiguration service to load the bitstreams specified by the `ID`.

VI. Results

An object tracking application was used to validate our approach. A video camera captures images and a set of operations is performed to track moving objects, which are then labeled and the result image is shown in a VGA screen.

The application is defined by 9 application components, with four main processing tasks: background substitution (BG subs), morphological transformation (morph trans), motion test and image update. Each one can use secondary tasks to perform compute intensive processing. Our reconfiguration design method is applied to the morphological transformation step. The morphological transformation task uses three auxiliary tasks: erosion, dilatation and reconstruction. They are

[1]Version 11 of Xilinx tools eliminates the need of BUS_MACRO, which simplifies this step. Our experiments are done in version 8.2i with EAPR (*Early Access Partial Reconfiguration)*

performed in sequence and the next one needs the preceding result to process data.

The platform contains 60 components, including 4 processors (3 microblaze processors and one PowerPC). The PowerPC which acts as the XManager apart from managing the whole system captures the next image and sends it to the first microblaze. The image processing is performed by the microblazes (XWorkers) in sequence, and the image update and display is performed by the PowerPC.

The microblaze processor that performs morphological transformation is connected to a co-processor (XModule) by two FSL busses, one to send request and one to receive results. Such a co-processor is dynamically reconfigured to compute the three image processing tasks (erosion, dilatation and reconstruction). Three IPs were generated: `copro_erosion`, `copro_dilatation` and `copro_reconstruction`, each one performs an image processing step belonging to the morphological transformation. For each IP a bitstream is generated and a fourth one, the empty bitstream, is also generated. Partial bitstream is about 80KB for each co-processor. The reconfiguration throughput is close to 80Mb/s, thus total reconfiguration time is about 2ms. In this case study we have considered the L1 level for the bitstreams repository as the global size of bitstreams does not exceed the SRAM size. As three reconfigurations are performed by the morphological transformation, 6ms is the total time overhead added by reconfigurability.

The time constraint for each step in the processing is 40 milliseconds (25 images processed per second). Each of the morphological transformation step takes less than 10ms, which leads to 36ms to perform the morphological transformation: 30ms for the computation and 6ms for the reconfigurations. In that case partial dynamic reconfiguration allows a significant area optimization (we used 1 co-processor instead of 3) without performance penalties. The necessary memory used by the reconfiguration service (XTask API) executed on the PowerPC is less than 80KB (40KB for executable code and 32KB for local data).

VII. Conclusion and future works

In this paper we have shown a global approach to design self-reconfigurable multiprocessor systems on chip. In order to provide a comprehensive approach we believe three key points must be addressed: i) definition of an efficient architectural model with adapted API in order to help designer during the design steps, ii) a bitstreams repository hierarchy to face potential huge number of bitstreams which will be required for future versatile systems and iii) a complete design methodology starting from a high level of specification (UML in our case). Increasing modeling abstraction levels allows to hide implementation details to the designer, leaving focus on system requirements rather than implementation issues.

In order to achieve our goals the platform must support dynamic and partial reconfiguration. We have applied our methodology to Xilinx Virtex series FPGAs. Although our experimentations are target-specific, our approach can be applied to other dynamic and partial reconfigurable platforms. Our approach allows a semi-automatic generation of dynamic reconfigurable MPSoPC. Results obtained demonstrated that replacing co-processors at run time allows area optimization without performance penalties.

References

[1] OMG, "The Unified Modeling Language (UML)," http://www.omg.org/uml. [Online]. Available: http://www.omg.org/uml

[2] D. et al., "Runtime adaptive multi-processor system-on-chip: RAMP-SoC," in *IPDPS*, April 2008, pp. 1–7.

[3] S. Vassiliadis, S. Wong, G. Gaydadjiev, K. Bertels, G. Kuzmanov, and E. Panainte, "The molen polymorphic processor," *IEEE Transactions on Computers*, pp. 1363–1375, 2004.

[4] M. Rashid, F. Ferrandi, K. Bertels, E. Informazione, and I. Milan, "hArtes design flow for heterogeneous platforms."

[5] A. Lagger, A. Upegui, E. Sanchez, and I. Gonzalez, "Self-reconfigurable pervasive platform for cryptographic application," *Field Programmable Logic and Applications, 2006. FPL '06. International Conference on*, pp. 1–4, Aug. 2006.

[6] J. W. Williams and N. Bergmann, "Embedded linux as a platform for dynamically self-reconfiguring systems-on-chip," in *ERSA*, T. P. Plaks, Ed. CSREA Press, 2004, pp. 163–169.

[7] J. Dekeyser, P. Boulet, P. Marquet, and S. Meftali, "Model driven engineering for soc co-design," *IEEE-NEWCAS Conference, 2005. The 3rd International*, pp. 21–25, 2005.

[8] Y. Zhu, Z. Sun, A. Maxiaguine, and W.-F. Wong, "Using uml 2.0 for system level design of real time soc platforms for stream processing," in *RTCSA '05: Proceedings of the 11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'05)*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 154–159.

[9] T. Wang, X.-G. Zhou, B. Zhou, L. Liang, and C.-L. Peng, "A MDA based SoC Modeling Apporach using UML and SystemC," in *Proceedings of the sixth IEEE International Conference on Computer and Information Technology (CIT'06)*, september 2006, pp. 245–245.

[10] P. Kukkala, J. Riihimaki, M. Hannikainen, T. D. Hamalainen, and K. Kronlof, "Uml 2.0 profile for embedded system design," in *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 710–715.

[11] E. Riccobene, P. Scandurra, A. Rosti, and S. Bocchio, "Designing a Unified Process for Embedded Systems," in *Fourth International Workshop on Model-Based Methodologies for Pervasive and Embedded Software*. IEEE Computer Science, mars 2007, pp. 77–90.

[12] S. Bocchio, A. Rosti, E. Riccobene, and P. Scandurra, "UML and MDA for Transactional Level Transactional Level Modeling," in *UML-SoC*, 2007.

[13] C.-H. Tseng and P.-A. Hsiung, "UML-Based Design Flow and Partitioning Methodology for Dynamically Reconfigurable Computing Systems." in *EUC*, 2005, pp. 479–488.

[14] Imran Rafiq Quadri, Samy Meftali, and Jean-Luc Dekeyser, "High level modeling of dynamic reconfigurable FPGAs," *International Journal of Reconfigurable Computing*, vol. 2009, p. 15, 2009.

[15] A. Rind, K. Shahzad, and M. Qadir, "Evaluation and comparison of tcp and udp over wired-cum-wireless lan," *Multitopic Conference, 2006. INMIC '06. IEEE*, pp. 337–342, Dec. 2006.

[16] N. Ploplys and A. Alleyne, "Udp network communications for distributed wireless control," *American Control Conference, 2003. Proceedings of the 2003*, vol. 4, pp. 3335–3340 vol.4, June 2003.

[17] N. Instruments, "Building networked applications with the labwindows /cvi udp support library," January 2009.

[18] A. Dunkels, "lwip," *Computer and Networks Architectures (CNA), Swedish Institute of Computer Science, http://www.sics.se/ãdam/lwip/*, 2001.

[19] J. Vidal, F. de Lamotte, G. Gogniat, J.-P. Diguet, and P. Soulard, "Ip reuse in an mda mpsopc co-design approach," in *ICM'09: Proceedings of the International Conference on Microeletronics*, 2009.