Hardware and software architecture facilitating the operation by the industry of dynamically adaptable heterogeneous embedded systems.

Laurent Gantel


ECOLE DOCTORALE SCIENCES ET INGENIERIE
Université de Cergy-Pontoise

PhD Thesis

Hardware and Software Architecture for Heterogeneous and Dynamically Reconfigurable Systems-on-Chip

by
Laurent Gantel

Equipes Traitement de l'Information et Systèmes (ETIS)
CNRS UMR 8051

Embedded System Lab (ESL)
THALES Research & Technology FRANCE

Thesis defended on 14^{th} January, 2014

M. Gilles Sassatelli	Reporter
M. Frédéric Petrot	Reporter
M. Daniel Chillet	Examiner
M. Guy Gogniat	Examiner
M. François Verdier	Director
M. Fabrice Lemonnier	Director
M. Mohamed El Amine Benkhelifa	Supervisor

Tell me and I forget, teach me and I may remember, involve me and I learn.

Benjamin Franklin

Abstract

This thesis aims to define software and hardware mechanisms helping in the management of the Dynamic and Heterogeneous Reconfigurable Systems-on-Chip *(DHRSoC)*. The heterogeneity is due to the presence of general processing units and reconfigurable IPs. Our objective is to provide to an application developer an abstracted view of this heterogeneity, regarding the task mapping on the available processing elements. First, we homogenize the user interface defining a hardware thread model. Then, we pursue with the homogenization of the hardware threads management. We implemented OS services permitting to save and restore a hardware thread context. Conception tools have also been developed in order to overcome the relocation issue. The last step consisted in extending the access to the distributed OS services to every thread running on the platform. This access is provided independently from the thread location and is is realized implementing the MRAPI API. With these three steps, we build a solid basis to provide to the developer in future work, a design flow dedicated to DHRSoC allowing to perform precise architectural space explorations. Finally, to validate these mechanisms, we realize a demonstration platform on a Virtex 5 FPGA running a dynamic tracking application.

Résumé

Cette thèse s'intéresse à la définition de mécanismes logiciels et matériels, facilitant la gestion des systèmes-sur-puce hétérogènes et dynamiquement reconfigurable *(DHRSoC)*. L'hétérogénéité de ses architectures se manifeste par la présence à la fois de processeurs de calcul généralistes et de modules matériels reconfigurables. Notre objectif est de permettre à un développeur d'application de s'abstraire de cette hétérogénéité en ce qui concerne l'allocation des tâches sur les différentes unités de calcul disponibles. Cette abstraction passe par une première phase d'homogénéisation des interfaces utilisateurs *(API)* et la définition d'un modèle de thread matériel. Cette homogénéisation se poursuit ensuite par la gestion de ces threads matériels. Nous avons implémenté des services au niveau du système d'exploitation *(OS)* permettant de sauvegarder et restaurer le contexte d'un thread matériel. Des outils de conception ont également été développés afin de surpasser le problème de la relocation d'un thread matériel au sein d'un FPGA. Enfin, la dernière étape a été d'étendre l'accès aux services offerts par tous les OS distribués au sein de la plateforme à tous les threads s'exécutant sur celle-ci, indépendamment de leur localisation. Ceci a été réalisé via une implémentation originale de l'API MRAPI. Avec ces trois étapes, nous avons apporté une base solide afin, dans le futur, de proposer au développeur un flot de conception dédié aux architectures DHRSoC lui permettant de procéder à une exploration architecturale précise de son système. Finalement, afin d'éprouver le fonctionnement de ces mécanismes, nous avons réalisé une plateforme de démonstration sur FPGA Virtex 5 mettant en scène une application de suivi de cibles dynamique.

Remerciements

Je voudrais tout d'abord remercier mes directeurs de thèse, Amine Benkhelifa qui m'a fait découvrir le monde de la recherche et m'a toujours poussé à aller plus loin, depuis mes premières années universitaires jusqu'au terme de ce doctorat, et qui a su me guider et me motiver tout au long de cette thèse, François Verdier dont les conseils et les remarques m'ont été utiles pour mener à bien ce projet, et Fabrice Lemonnier qui m'a fait confiance et m'a accueilli au sein du laboratoire LSE chez Thales Research and Technology durant mon Master et ma thèse.

Merci également aux membres du jury qui m'ont fait l'honneur d'évaluer mon travail, Gilles Sassatelli et Frédéric Petrot qui ont accepté d'en être les rapporteurs, Daniel Chillet et Guy Gognat qui en ont été les examinateurs.

Je tiens en particulier à remercier mes collègues de bureau, Amel Khiar, qui a toujours été là pour m'encourager et avec qui j'ai passé d'excellents moments. Je la remercie encore pour sa bonne humeur communicative et tout ce qu'elle m'a apporté durant toutes ces années. Un grand merci à Liang Zhou que j'ai appris à connaître et à grandement apprécier au fil du temps. Merci également à Lounis Zerioul, Guy Wassi, et Christian Gamom, qui ont aussi été très présents et qui sont devenus au fil du temps de véritables amis.

J'adresse mes remerciements aux membres de TRT que j'ai eu la chance de cotoyer, avec lesquels j'ai pu collaborer dans un environnement de travail agréable, et dont les diverses compétences m'ont été très utiles et surtout très instructives, parmi lesquels Jimmy Le Rhun, Christophe Clienti, Paul Brelet, Rémi Barrere, Téodora Petrisor, Philippe Millet, Philippe Bonnot et Lionel Thavot, ainsi qu'aux membres du laboratoire ETIS dont entre autres Frédéric de Melo, Lounis Kessal, Emmanuel Huck, Samuel Garcia, Thomas Lefebvre, Kaouthar Bousselam, Laurent Rodriguez, Benoit Miramond, Lotfi Bendaouia et Fakhreddine Ghaffari.

Une part de ces remerciements va aux membres du projet FOSFOR avec lesquels j'ai travaillé régulièrement: Fabrice Muller, Daniel Chillet, Sébastien Pillement et Nicolas Knecht.

Enfin je souhaite exprimer toute ma gratitude envers ma famille et mes proches pour leur présence et leur soutien durant toutes ces années.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Contents

1.1 Context

1.1.1 Real-time applications for embedded systems

Applications for embedded systems dedicated to image and signal processing are becoming increasingly complex. The amount of data processed by these systems tend to be more and more important and so, developers need more and more computing power. This is the case for instance, of monitoring system, automotive or radar applications. This leads to design new computing systems able to respect the high performance constraints imposed by these applications and their environment.

In order to satisfy these constraints, applications must be profiled and divided into several tasks. Each task which is considered responsible for the failure to hold constraints, has to be implemented separately on a dedicated processing unit. For instance, communicating systems such as a network switch, have to handle several protocols, transfer information at high rates and process large amount of data. To achieve good performances and gain in flexibility, communication protocol stacks may be implemented in hardware and take advantages of the partial and dynamic reconfiguration *(Fig. 1.1)*.

In general, the multiplicity of features needed by the end-users and mostly the specificity of these features, force designers to propose new architectures. Targeting heterogeneous processing units to deploy an application allows to accelerate the

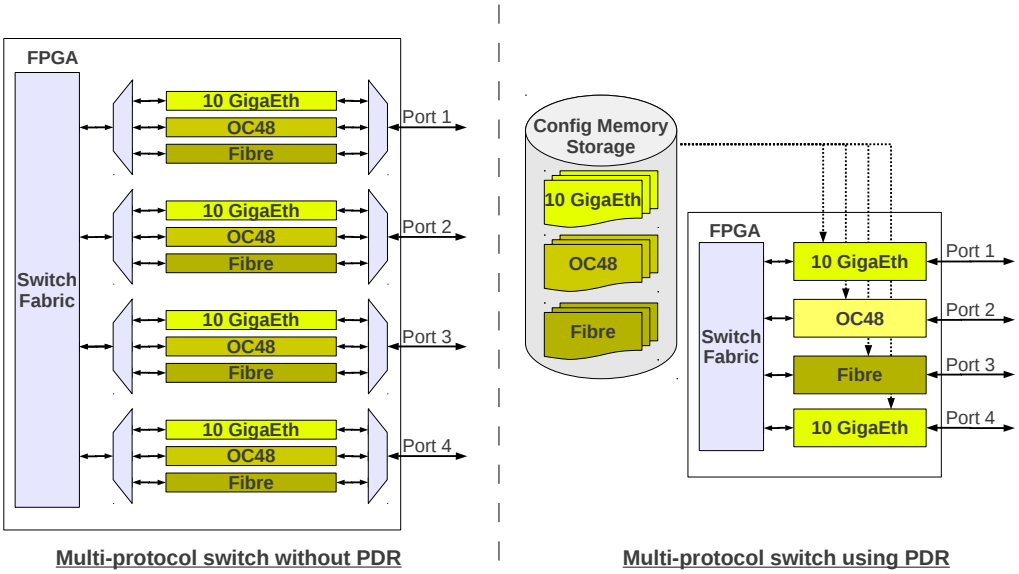


Figure 1.1: Partial and Dynamic Reconfiguration *(PDR)* application example [Xilinx 2010a]

global performance of the application. However, the drawback is that it complicates the development process.

Another constraint is the need of flexibility, or more precisely, of adaptability. The applications complexity requests to adapt the parameters and the provided features of these systems. For example, the computation power can depend on the quality of service required, and the power consumption of a system can be monitored regarding its environment or random events. Also, as embedded systems are more and more integrated in our environment, these human or environmental interactions require these systems to adapt themselves to the various queries and needs that this implies.

In contrast, designers would want to get a simple view of their application which would abstract the platform specificity, especially the heterogeneity *(Fig. 1.2)*. The aim is to dissociate the functional validation of the application and the design exploration of its implementation.

In the functional validation, tasks are described regarding high-level execution parameters such as the execution time, the deadline, or the priority. During the design exploration, these parameters and new ones like the power consumption or the memory usage are added regarding one or several possible partitioning. These two points lead us to consider the design of heterogeneous systems-on-chip and the way we can leverage the issue of their programming complexity in order to ease the move from the high-level modelling layer to the physical implementation.

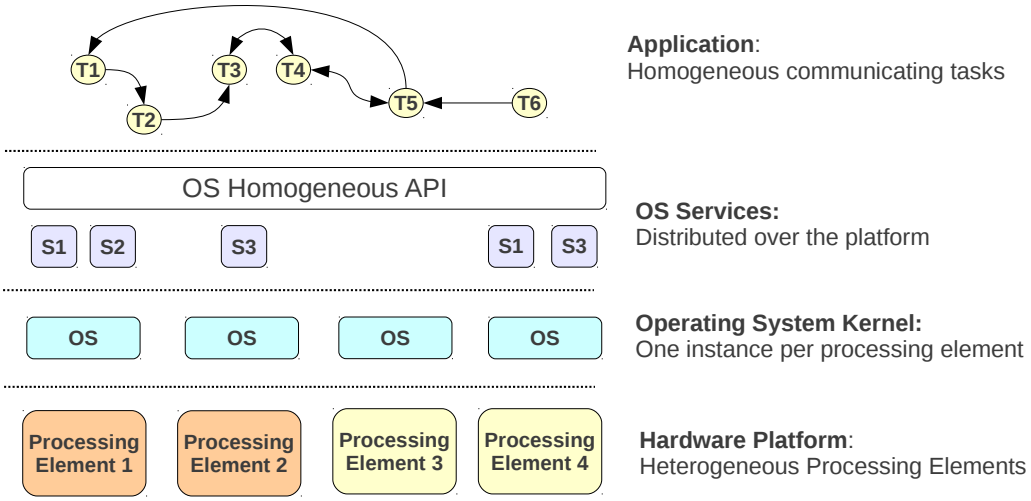


Figure 1.2: Design flow from developer's point of view

1.1.2 Heterogeneous Systems-on-Chip

Platforms based on different processing elements are called Heterogeneous Systems-on-Chip *(HSoC)*. In such a platform, the application is divided into tasks. Whereas some tasks are implemented as hardware accelerators and allocated into a partition of the chip, others run as software tasks on computing processor elements. A hardware accelerator is defined as a hard-wired function developed to accelerate the processing of a task. A computing processor unit could be a General Purpose Processor *(GPP)*, a specialized one like a Digital Signal Processor *(DSP)*, a Graphics Processing Unit *(GPU)* or a simple Micro-Controller Unit *(MCU)*.

Each one of these processing elements is more or less suited to certain types of tasks [Leon Adams 2007]. The hardware accelerator is well suited to intensive processing tasks, especially tasks whose operations can be parallelized. On the contrary, it can hardly be used with intensive control tasks. The latter are more suitable to run on a GPP. Homogeneous tasks with a low data dependency can be easily and efficiently parallelized on a GPU, whereas heterogeneous tasks with complex data paths are not recommended for this architecture. Simple control tasks processing small and well ordered data would likely be implemented on a Micro-Controller Unit. Playing with these different processing elements, it is possible to adapt the application to be deployed regarding time execution constraints or memory and logic resources. Table 1.1 summarizes strengths and weaknesses of each processing element.

Like it is possible to execute a task on different type of processing units, the

Processing Element	*Control*	*Performances*	*Programmability*
GPP	+++	+	+++
GPU	+	++	++
DSP	+	++	++
MCU	+++	+	+++
Hw. Acc.	+	+++	+

Table 1.1: Processing Elements comparison regarding control ability, performances and general programmability

platform which includes all these components can be implemented using different technologies: an Application Specific Integrated Circuits *(ASIC)*, a Multi-Processor System-on-Chip *(MPSoC)*, or a Field Programmable Gate-Array *(FPGA)*.

ASIC technology offers great performances but is very expensive and not flexible at all. In this document we consider a MPSoC as a SoC made up of at most a dozen of cores like the OMAP5430 based on a Cortex-A15 multiprocessor core [Instrument 2011]. They are less efficient but cheaper, more flexible regarding tasks placement and software bugs may be recovered. FPGAs is a good trade-off between the ASIC technology and the MPSoC choice because it is flexible, it provides better performances compared with MPSoC and both software and hardware bugs may be recovered after the application system being placed on the market. Table 1.2 summarizes the strengths and weaknesses of each technology.

Technology	*Cost*	*Flexibility*	*Performances*
ASIC	+	+	+++
FPGA	++	+++	++
MPSoC	+++	++	+

Table 1.2: Platform technology comparison regarding control cost, flexibility and performances

The solution which interest us is the FPGA technology. The exact reasons of this choice, namely the characteristics, the potential as well as the pros and cons of the last family of FPGA are detailed in the next subsection.

1.1.3 Modern FPGAs

A FPGA is a reconfigurable chip composed of several logic elements whose the configuration, that is to say the logical function they implement, as well as the interconnections between them can be modified on the user's willing.

A modern FPGA is a matrix of resources disposed in parallel columns. Each column contains either configurable logic blocks *(CLB)*, but also block ram memories *(BRAM)* or dedicated digital signal processing *(DSP)* blocks. For this platform we defined a hardware accelerator as hard-wired function using a set of resources allocated in a partition of the FPGA.

In addition to these configurable elements, latest families of FPGAs, for instance Xilinx Virtex 7 FPGAs *(Fig. 1.3)*, include hardcore elements to accelerate certain processing or communication. This is the case of the DDR controller, the Ethernet MAC controller, or even of hardcore processors implemented with all the needed peripherals as a full micro-controller unit *(dual ARM9 cores with timers, UART, or ICAP (Internal Configuration Access Port) controllers)*.

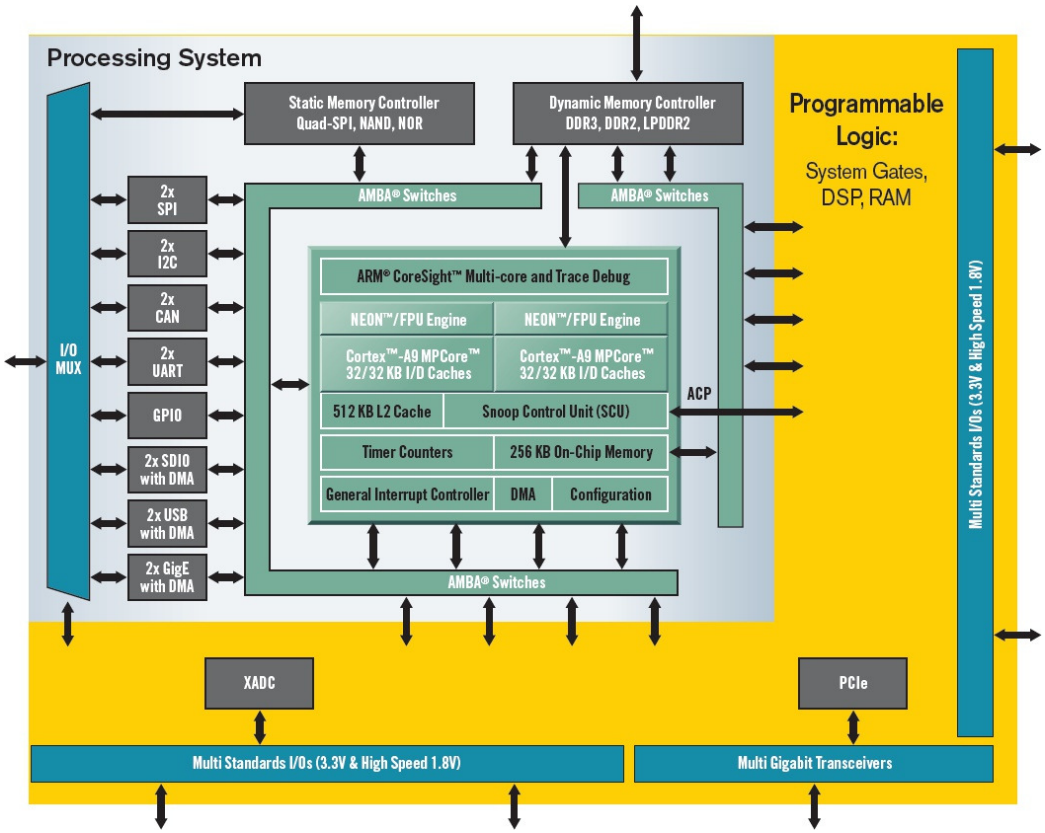


Figure 1.3: Xilinx Zynq 7000 EPP block diagram

As modern FPGAs matrices tend to become larger and larger, designers have now more space to implement multi-core systems including several soft-processors and hardware accelerators. In order to offer the best performances, and as told previously, the use of FPGA as a complete autonomous system is becoming a good trade-off between the ASIC technology and the multiprocessor solution. The first

is really efficient but rather expansive for small production lines, whereas the latter is flexible and can relies on many COTS but doesn't allow to reach the wanted performances. Namely, a FPGA is a good trade-off between power consumption and processing power.

Moreover, all processing units detailed in Section 1.1.2 can be implemented inside a FPGA. This capability provides to the developer the flexibility to explore different solutions when designing his platform. Several architecture choices can be made and compared. Each function can then be implemented on the wanted processing units in order to obtain the best partitioning.

1.1.4 Dynamic and Partial Reconfiguration

The natural evolution of FPGAs leads them, due to the miniaturization, to offer more and more logic resources [Koch 2010b]. This increase helps to face the important need of features required by the end-user. To manage the dramatic increase of the size of the FPGAs, especially the design time, manufacturers provided partial reconfiguration features to their FPGAs *(Fig. 1.4)*.

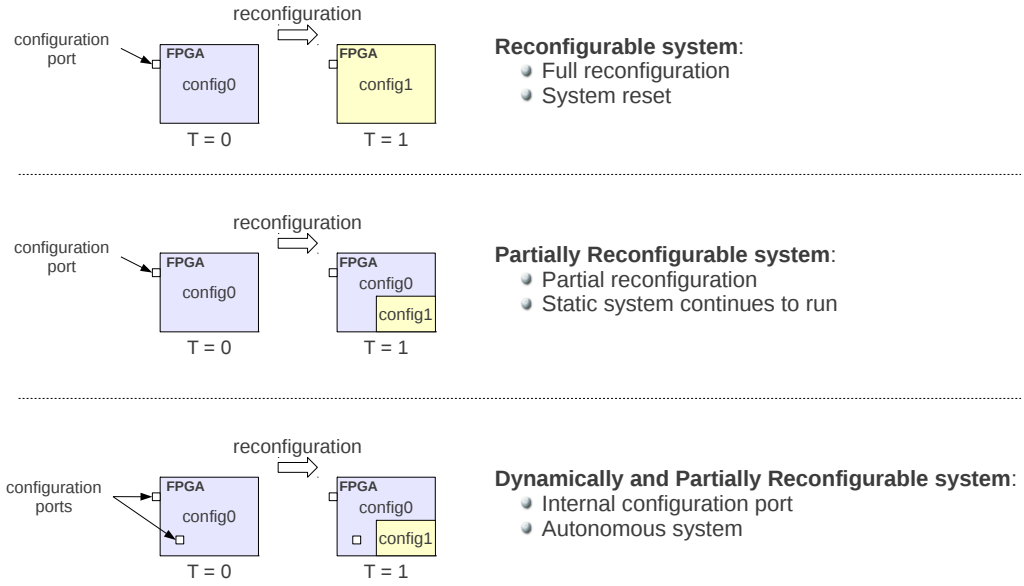


Figure 1.4: Dynamic and Partial Reconfiguration principle

The use of the partial reconfiguration has the advantage of decreasing the implementation time because partial modules can be implemented separately while the static part of the system remains the same and so do not need to be reimplemented.

Modern FPGAs manufacturers, from now Xilinx and Altera, provide some mechanisms to dynamically reconfigure the chip. The dynamic reconfiguration allows to reconfigure a partial module while keeping the static part unchanged. The system on the chip would be able to reconfigure a part of itself, this without any disturbance on the execution of the rest of the system. In addition to the functional interest, it brings a consequent resources impact for autonomous embedded systems-on-chip. Moreover, in some cases it is a good way to decrease the power consumption while being capable of providing a larger choice of hardware accelerators to a given application.

1.2 HSoC programming model

1.2.1 Programming issue

Despite the real interest of this technology, the main drawback of using heterogeneous platforms is that they are difficult to program. Indeed, abstraction level differences between software functions running on processors and hardware accelerators, make the development of applications really tough. In order to ease the validation and the exploration of the possible partitioning for a given platform, a common abstraction has to be provided to the end-user *(Fig. 1.5)*.

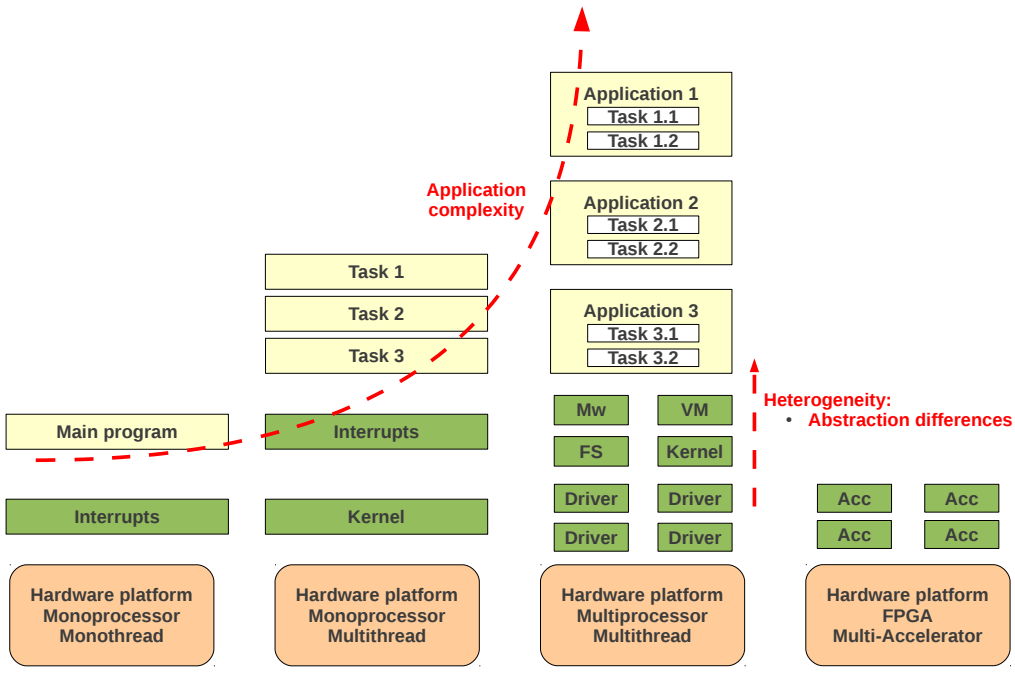


Figure 1.5: Abstraction level differences between hardware and software programming models

To achieve it, a general trend which is emerging consists in adopting a high-level language to describe the application. Coupled with new efficient tools able to simulate and automatically generate low-level code sources, such a design flow would allow to tackle the last FPGAs programming issues. Indeed, due to their increasing size, the system complexity is increasing too and such tools would provide a simpler view of the whole system. For instance, a language such as the Synchronous Data-Flow language *(SDF)* [Lee 1987] provides a model of computation which can be adapted both to software and hardware threads, and so abstract the heterogeneity of the platform.

An intermediate approach can be adopted which provides not a unique programming language to describe both the software and the hardware, but in a first step, a common programming model. In this way, a commonly adopted programming model in the software embedded domain is the threading model. To design a heterogeneous platform using this model, we have to raise the abstraction level of the hardware accelerators. This allows us to reuse legacy works in the software domain and so to focus on the hardware part of the model. In our case, the implementation choice is done between a software implementation on a processor and a reconfigurable hardware logic partition.

Like software threads, we define hardware threads. A hardware thread encapsulates the hardware accelerator and allows it to behave like a software thread. Namely, a hardware thread would be able to access operating system services and would have, from a certain point of view, a sequential execution. These services include the ability to create or delete a resource, and to operate a system call. The user should have the capability to preempt any thread, both software or hardware, and so to save and restore its context. A particular effort should be done on the implementation of mechanisms permitting the threads to communicate in a transparent way. Our final objective is to offer to the end-user a simple thread view of its application, and to the designer an efficient way to create relocatable hardware accelerators which act like software threads *(Fig. 1.6)*. To do so, hardware accelerators should be implemented in what we will call a hardware thread to communicate.

Developed in the standard hardware description language which is VHDL *(Very High-speed integrating circuit Development Language)*, generic interfaces and an abstracted execution model will allow in the future to integrate this intermediate programming model with high-level design tools. This will result in the automatic generation of hardware threads, taking advantage of existing low-level structure.

1.2.2 Dynamically Reconfigurable HSoC

In this context, the dynamic and partial reconfiguration of FPGAs seems very interesting to provide a flexible handling of hardware accelerators. In dynamically

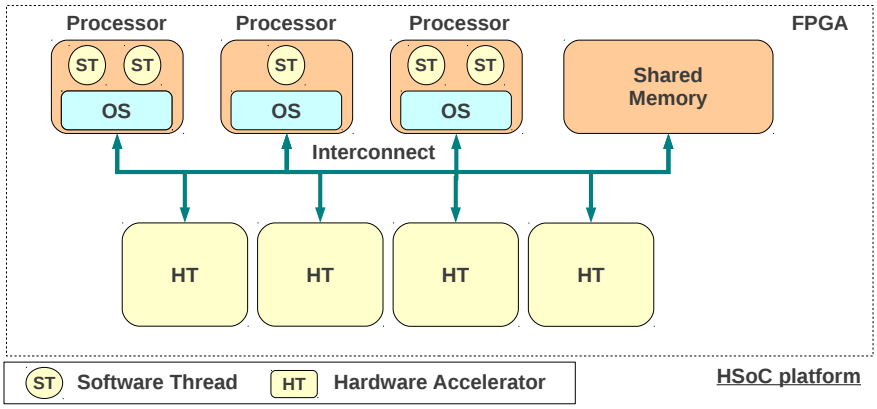


Figure 1.6: Heterogeneous threading application

reconfigurable HSoC, hardware threads are defined as relocatable modules which can then be allocated into any available reconfigurable partition of the FPGA.

The system, using the dynamic and partial reconfiguration, allows the user to preempt any module *(Fig. 1.7)*. Namely, a part of the chip is divided into several dynamic partitions. Each partition is then allocated by the control part of the application to one hardware thread for a certain amount of time.

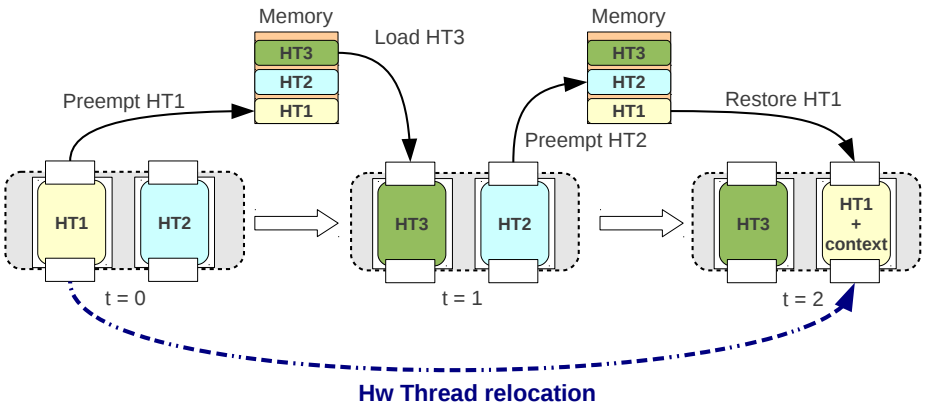


Figure 1.7: Hardware Thread preemption

The list of target applications can then be extended to multi-mode applications and to those which need environment adaptation. As a perspective, other applications based on the dynamic detection of events, such as security systems, could take advantage of this technology. We can also cite bio-inspired architectures which would rely on dynamic reconfiguration mechanisms in order to dynamically reconfigure their architecture.

Moreover, being able to update system after its release could help the designer to improve the adaptability to unknown specification modifications, for instance when implementing a H264 codec. We can also notice that it could have a good affect on the design costs of these products.

1.3 Objectives

The goal of this PhD thesis is to propose a software and hardware architecture in order to improve the application development process when targeting a Heterogeneous System-on-Chip. With the increasing complexity of the application, an abstracted programming model has to be adopted to facilitate the description of these applications and improve the flexibility regarding the implementation choices. The proposed architecture should rely on the existing operating system structure and provide services and low-level mechanisms to easily handle the thread heterogeneity.

In Chapter 2, we propose a model of hardware thread which allows to abstract this heterogeneity. Then we study mechanisms and tools permitting to manage hardware threads in the same way that what is done with software ones. In the next chapter, an operating system dedicated to heterogeneous systems-on-chip is specified. The main feature of this operating system is to provide a flexible access to the operating system services for every threads, both software or hardware, whatever is the core they are running on. Finally, an application will be detailed and implemented on a demonstration platform.

Chapter 2

Unified Thread Model

Contents

2.1 Related work

2.1.1 Software kernel management

With the emergence of heterogeneous platform including both software processors and reconfigurable areas, a natural way to tackle the heterogeneity of these reconfigurable platforms has been to rely on the existing software abstraction layers. To rise their abstraction level, the control of the hardware accelerators has been given to a software operating system running on a processor.

This scheme leads to design a new kind of platform in which a primitive or function used by a task, or a task itself, can be accelerated in hardware. The following works aim to provide a simple way to load and run these accelerators. They permit to abstract the complexity of the communication between a processor, namely an application running on top of an operating system, and a hardware accelerator.

This is the case of the Egret platform [Bergmann 2003] which the objective is to provide a fully modular platform. A Microblaze micro-controller unit is running a μC-Linux operating system and allows the developer to choose which hardware accelerators have to be executed. To do so, a classic driver using the IOCTL[1] API[2] [IOC 1997] permits the developer to load a partial bitstream of the wished configuration through the Internal Configuration Access Port *(ICAP)* of the FPGA *(Fig. 2.1)*.

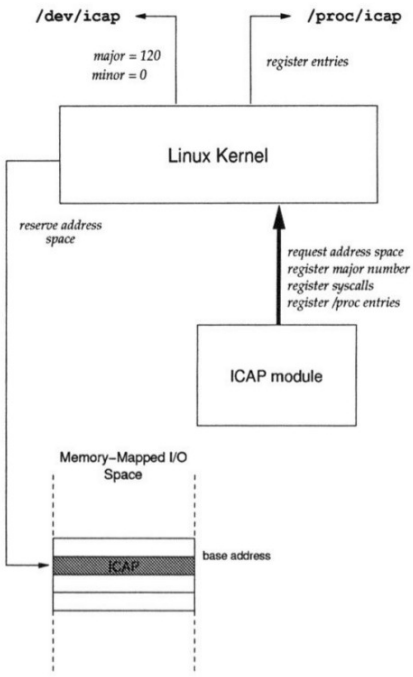


Figure 2.1: μC-Linux ICAP driver [Bergmann 2003]

Authors of [Donato 2005] presented a platform based on the Linux operating system. This choice has been done because its source code is available for free, it has been ported on numerous platforms and it is modular regarding additional drivers.

This platform has been named Caronte : it is composed of a Virtex 2 Pro FPGA

[1] Input and Output Control
[2] Application Programming Interface

including a Power PC 405 and one ICAP port. A software driver allows the developer to control the ICAP using the IOCTL protocol again. When loading a new IP[3] core, a communication protocol has been implemented to allow this IP to claim itself to the Core Manager IP, following a hot-plug philosophy. The interconnect is a Wishbone Bus and specific Medium Access Controller *(MAC)* are used to provide the ability to allocate address space at run-time. This work led to the launch of the commercial project *PetaLinux*, which aims to simplify the deployment of the Linux operating system on reconfigurable platforms. The use of Linux in MPSoC platforms is a growing trend as shown by the recent acquisition of the PetaLogix company by Xilinx.

In [Rana 2007], a platform composed of several FPGAs is introduced. The whole platform is supervised by a unique processor running Linux, and allowing reconfiguration ability, partially or totally. Simple primitives are also implemented as a driver using the IOCTL protocol.

The main issue to solve is the management of the concurrent execution of each task present in the system. To handle this, we need to rely on a multitask operating system providing simple and legacy ways of communication to every task, both software or hardware *(Fig. 2.2)*. Especially, hardware tasks are connected to a Medium Access Controller *(MAC)*, which provides the ability to dynamically allocate address space for each loaded module at run-time.

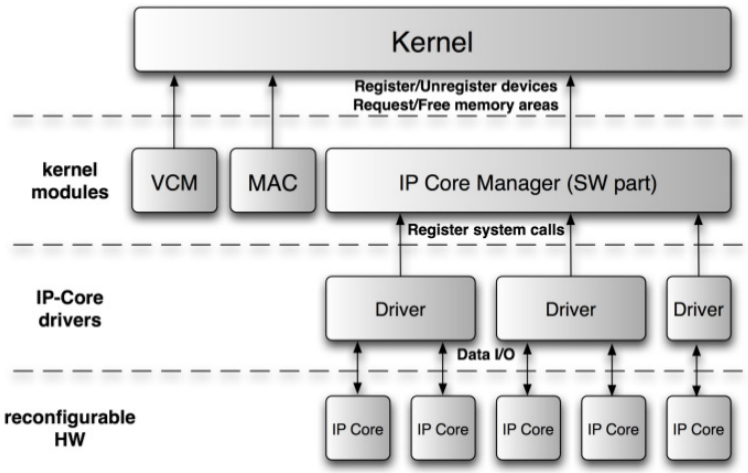


Figure 2.2: RAPTOR software architecture [Rana 2007]

The operating system used to abstract the reconfiguration process is based on the work of Donato et al. [Donato 2005]. When a reconfigurable accelerator is loaded on the FPGA, a driver is loaded into the Linux kernel and is associated to this accelera-

[3] Intellectual Property

tor. To control the module, the application relies on the classical IOCTL commands.

In all these works, the management of the hardware accelerators implies minimal modification in the operating system and is easily portable. However, the accelerator is considered as a hardware IP core and not as a hardware thread. From the user point of view, this situation leads to a heterogeneous programming model for the developer. It is not sufficient regarding our objectives which impose us to bear in mind to allow a homogeneous programming model at a higher level of representation.

2.1.2 Run-time manager

Other solutions go further and propose to design a run-time manager. A run-time manager is responsible for scheduling hardware accelerators at run-time and managing the access to shared resources. The system knows which partitions are available and which accelerators need to be loaded. Using adaptive algorithm, a real-time unit dynamically places and configures the accelerators. More than a management of the hardware accelerators as co-processor modules, the goal is to define a model in which these accelerators could be considered as real tasks, in the same way that the software ones are.

Nollet et al. [Nollet 2003] introduces one of the first approach to design an operating system dedicated to Reconfigurable Systems called OS4RS. It specifically targets the Heterogeneous Reconfigurable System-on-Chips composed of ISP *(Instruction Set Processor)* and reconfigurable tiles.

This OS must be capable of providing a similar set of services for the heterogeneous tasks, as a traditional OS does for software application. It is based on RTAI, a real-time Linux extension.

The hardware task are placed into slots and connected to each other via a network-on-chip. The Hardware Abstraction Layer *(HAL)* of the operating system provides communication primitives such as send and receive as well as control messages to place a new task and read or modify the network parameters *(Fig. 2.3)*. The communication API has been ported both in hardware and software. This common interface allows to migrate a task from a software to a hardware processing element in a transparent way.

The operating system includes a two-level scheduler. The first level dispatches the task on the processing units whereas local schedulers handles the task assigned to them. At the first level, the scheduler relies on a checkpointing mechanism to save tasks contexts. They choose this solution because this has the advantage to make the context independent from the targeted processing element. A the lower level, local schedulers may employ processor-specific contexts, since they will never move tasks to another processor. The definition and the management of the checkpoints *(ie. the definition of what needs to be saved)* is up to the user. We can notice that this information is particularly difficult to define and is still an open issue.

In addition to the scheduling service, the operating system provides a relocation service using the checkpointing mechanisms to synchronize the migration.

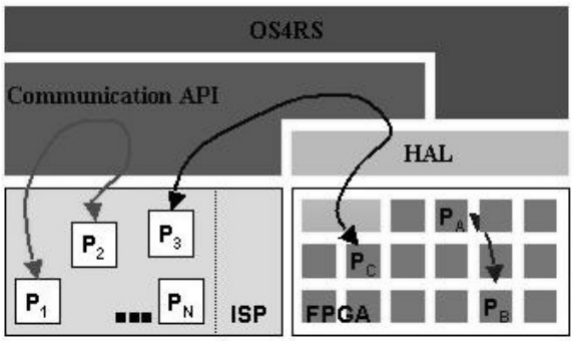


Figure 2.3: OS4RS platform architecture [Nollet 2003]

In [Steiger 2004], Platzner et al. also introduce an operating system dedicated to reconfigurable systems and discuss about two different points. The first discussion is about design issues for reconfigurable hardware operating system. The required degree of flexibility paired with high computation demands asks for partially reconfigurable hardware that is operated in a true multitasking manner.

For the authors, it is necessary to define three things: (1) a programming model dedicated to reconfigurable systems with a set of well-defined system services, (2) a run-time system to handle the dynamicity of the system and resolve conflicts between executable objects, and (3) the smallest unit of execution, that is to say a process or a thread.

They define a hardware thread as a pre-placed and pre-routed digital circuit which can be loaded and relocated easily in any available slots of the FPGA. A square is the simplest shape to manage in spite of the fact that it also leads to a more important internal fragmentation than more complex shapes, such as polyominoes. Then they explain that 1-Dimensional *(1D)* placement involves an easier scheduling of the different threads but an increase of the external fragmentation. On the other hand, 2-Dimensional *(2D)* placement offers more possibility of placement and so less external fragmentation but is harder to manage.

In this paper, they target a real-time scenario where each incoming thread is either accepted with a guarantee to meet the deadline or rejected. As in reality FPGA resources distribution is not homogeneous, we can assume that at least memory and FIFOs are managed by the operating system, and so that a thread can access to these resources using operating system services : memory allocation and message queue. They conclude saying that 1D placement is more realistic regarding current FPGAs architecture but 2D placement is an interesting open issues in the way that 2D scheduling is really more interesting in term of performance.

The second discussion deals with hard real-time tasks scheduling. Target platform is composed of a CPU connected to a reconfigurable device through two ports: a C/R port for configuration and readback, and a COMM port for communication

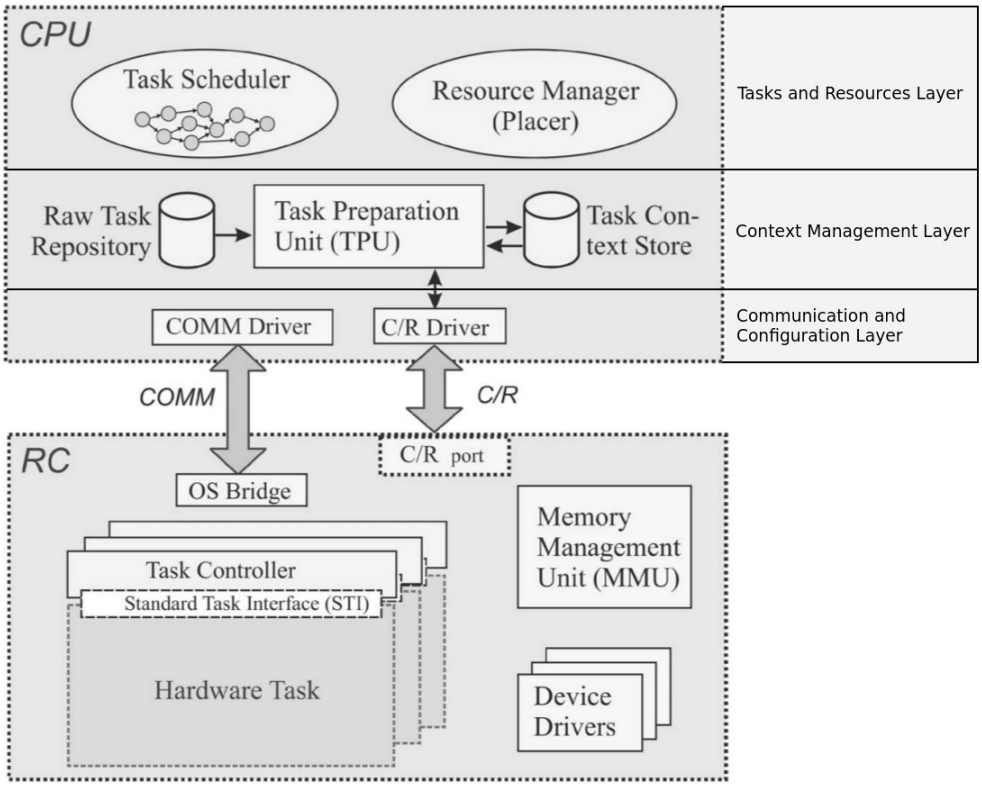


Figure 2.4: Operating System for Reconfigurable Systems software architecture [Steiger 2004]

between the operating system and the thread. This port is called Standard Task Interface *(STI)*. The Task Communication Bus *(TCB)* runs horizontally through all hardware thread area into a number of dummy tasks.

The software operating system is divided in three layers *(Fig. 2.4)*: a first layer to manage tasks and resources, a second to handle the context issue, and the last one which is responsible for the communication and the configuration.

In [Wigley 2001], authors discuss the scheduling problem of relocatable hardware tasks by an operating system. They give a specification of an ideal operating system dedicated to the reconfigurable computers. This operating system must provide a scheduler able to manage explicit context changes, namely the user has to insert checkpoints inside tasks source code in order to ensure a correct context save.

In their specification, the operating system is responsible for managing the virtual memory and protecting platform physical resources from conflicting accesses. Task partitioning must be dynamic as we must be able to operate load balancing or task migration from software to hardware and vice-versa.

Also, communication between hardware tasks must be thought in order to be optimized. If two tasks are presents on different slots, we must take advantage of

it by initializing direct communication between these kind of tasks. Otherwise, a buffer should be used in order to process communication. A last point is the need of verification tools and test cases, that is to say application examples which could benefit from the Dynamic and Partial Reconfiguration.

Another example of run-time manager is introduced in [Shiyanovskii 2009b]. Reconfiguration is managed by a software layer upon the real time operating system. This layer is called *Adaptation Manager*, and can be customized in order to get a trade-off between the power consumption and the execution speed. To do so it relies on a learning process which allows it to improve its decision skill.

The reconfigurable platform is composed of tiles which abstract the logic block programming level to provide to the developer an access to coarse grain primitives such as filters, FFT[4] or others higher level functions. Scheduler policy is based on priority. Tasks can have three different states : Inactive, Active and Reserved and have real-time attributes such as execution time, deadline, or laxity.

These works show that an operating system is necessary to manage the hardware accelerators. This abstraction layer has to take advantage of the dynamic reconfiguration and provides high-level mechanisms to manage the available slots. It means offering the ability to the end-user to create, suspend, resume and delete a hardware task. At a lower-level, a reconfigurable partition should be seen as a processing element. The operating system should be able to share this resource between every hardware accelerators, leading us to view a hardware accelerator as an equivalent of a software thread.

2.1.3 Hardware thread model

Using the ability to control the Dynamic and Partial Reconfiguration *(DPR)*, recent articles proposed abstraction models for the hardware accelerators. The objective is to improve the programmability of these heterogeneous platform and to facilitate the communication between the accelerators and the rest of the system providing a default interface.

Authors of [El-Araby 2008] define VFPGAs. This acronym stands for Virtual FPGAs. A VFPGA is a reconfigurable zone controlled by a processor *(Fig. 2.5)*. A VFPGA can be seen as a hardware task. This kind of task has three different states: configured and waiting for input data *(data in)*, processing, or sending data *(data out)*.

A virtualization manager is implemented to receive execution requests coming from processors. It is responsible of loading the VFPGAs. As expected, different tests show a gain regarding the execution speed.

[4] Fast Fourier Transform

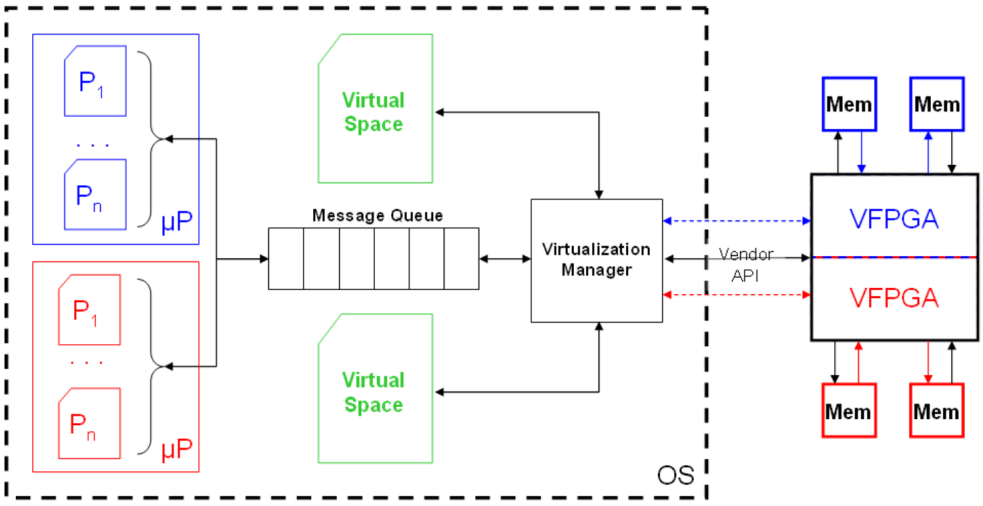


Fig. 5. Run-Time virtualization layer using PRTR

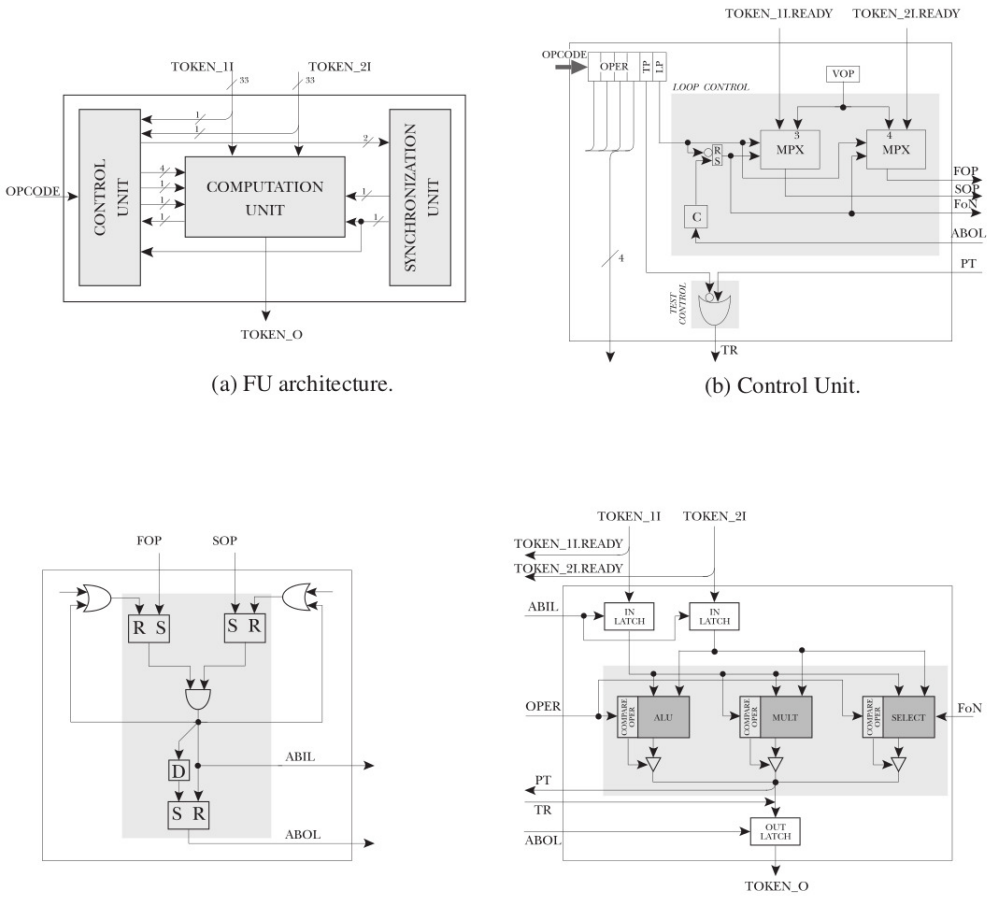
Figure 2.5: VFPGA runtime manager architecture [El-Araby 2008]

In [Verdoscia 1994], authors tackle the issue of the hardware implementation of a Data-Flow Graph *(DFG)* model of computation *(MoC)*. In a DFG model, a process can be represented by an actor. Actors communicate by sending each other packets of data called tokens [Lee 1987]. Although this model is generally static, this paper defines a dynamic model in which actors inputs and outputs tokens come and go from and to infinite FIFOs. Every actors have two inputs and a unique output allowing to define three types of links between them:

- classical link: $2 \rightarrow 2$ *(two outputs of two different actors to the inputs of one or two other actors)*
- joint link: $2 \rightarrow 1$ *(two outputs of two different actors to the inputs of another actor)*
- and replica link: $1 \rightarrow 2$ *(one output of an actor to the inputs of one or two other actors)*

Actors are grouped in clusters which communicate by Message Passing. Inside a cluster, actors are called Functional Units *(FUs)*. These FUs communicate through a crossbar. Messages exchanged between FUs and between FUs and the host correspond to the graph configuration and the produced tokens. A FU is composed of three elements *(Fig. 2.6)*:

- "Control Unit": this component permits to manage loops and conditions, this using Test Macro



(a) FU architecture. (b) Control Unit.

(c) Synchronization Unit. (d) Computation Unit.

Figure 7. An FU design

Figure 2.6: Functional Unit architecture [Verdoscia 1994]

- "Synchronization Unit": it is responsible for controlling the presence of the input tokens. Two signals are generated: ABIL if the two tokens are present, ABOL with a delay of one cycle to allow output firing
- "Computation Unit": it composed of an ALU [5], a multiplier and one Selection module. If a test is requested and that it passes, the output is activated on the arrival of the ABOL signal

The proposed model has three advantages. Firstly, all actors have the same architecture *(two inputs - one input)* so the same interfaces with the external world, then it allows to get an architecture adapted to VLSI, and finally all actors are able to manage loops and conditional instructions.

[5] Arithmetic and Logical Unit

Much more complex accelerators have then been developed, such as Hybrid Thread [Agron 2009a]. In this article, the authors define a model of POSIX[6] compliant hardware thread, capable of processing operating system calls through a shared memory, as software thread does *(Fig. 2.7)*. A thread is composed of two finite states machines *(FSM)*. One used to answer to operating system requests and get system calls results, and the other one to process system calls and get access to a heap. These FSMs are controlled by the hardware accelerators encapsulated in the User Logic component.

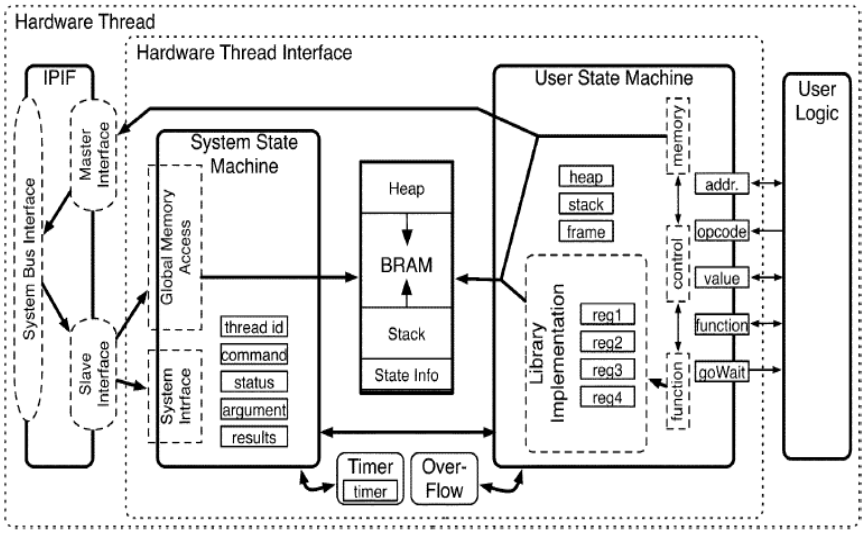


Figure 2.7: Hybrid Thread model [Agron 2009a]

Heap and stack are stored in an internal Block RAM *(BRAM)* of the thread. Like in a software POSIX thread, the stack is used to store the system calls parameters. Moreover, in order to enhance the programmability of these threads, the authors defined a high-level API which allows the developer to describe a heterogeneous application using the C language. A dedicated compiler written in Python permits to translate the C code into a VHDL implementation of the Hybrid Thread.

In [Lubbers 2008], the authors introduce an operating system dedicated to reconfigurable architectures: ReconOS. This operating system provides a homogeneous abstraction layer to the threads, both software or hardware, and allows them to process system calls. This paper deals with the portage of ReconOS on a Linux based platform, and compares its performances with another one based on the eCOS operating system. The goal is to demonstrate the portability of the concepts brought by the ReconOS architecture.

[6] Portable Operating System Interface

In this operating system, every services are managed by a software operating system running on a processor. Hardware system calls are done through an API described in a VHDL library. The hardware thread finite state machine is synchronized with the software operating system in order it to process the system call. The interface responsible for the communication is called OSIF for OS InterFace and represents a set of registers accessible through the processor bus *(Fig. 2.8)*.

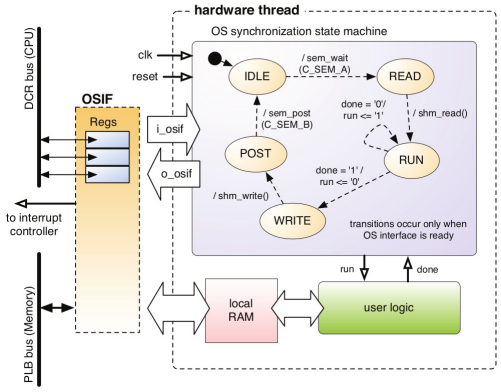


Figure 2.8: ReconOS hardware thread model [Lubbers 2008]

Regarding the inter-thread communication, the thread heterogeneity is abstracted associating each hardware thread with a software one, which is a proxy or a delegate. When requested by a hardware thread, the system call is executed by the corresponding software thread.

In order to link operating resources requested by the hardware thread with the ones accessible by the software one, a table of the used instances is maintained by the delegate. In this way, the same hardware thread can be used by several instances of a software thread. This mechanism has been implemented to foresee the future use of the partial and dynamic reconfiguration.

2.1.4 Conclusion

As explained in the introduction, our choice is oriented to the threading model. Our goal is to propose a hardware thread model which is able to communicate with software threads in the same way that what has been proposed by Hybrid Thread [Agron 2009a] or ReconOS [Lubbers 2008]. This model has to be adapted to the reconfigurable platform and take advantage of the parallelism and the flexibility offered by this type of platform. The definition of this model is the basic proposal of this thesis and will lead us to define in the next chapters, an operating system architecture which offer the ability to abstract the specificity of the hardware thread regarding the software one.

2.2 Thread model

2.2.1 Process definition

A process is defined as an independent stream of instructions, running on top of a processing element. A process permits to group some of a processing element resources together, such as the memory space, the open files, the signal handlers and other information. Grouping resources inside a same entity facilitates the management of these resources by the running process [Tanenbaum 2001].

Process execution is protected by the fact that it has a private address space. Processes are scheduled by the kernel operating system and compete for the access to the processing element. When a process is blocked by a system call, the scheduler is responsible for saving the context of this process and selecting another process among the ones ready to be executed.

2.2.2 Thread definition

A thread is executed inside a process *(Fig. 2.9)*. The main difference between a thread and a process is that the latter has a full view of the memory space addressable by the processor whereas threads inside a same process share the processing element resources owned by the process.

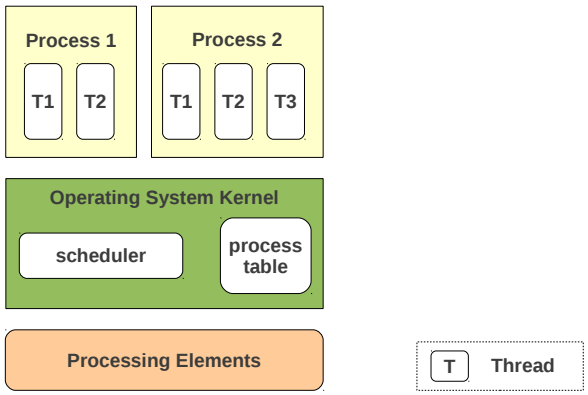


Figure 2.9: Process and Thread

A threading model provides the advantage to isolate application functions executions regarding one to the others and so enforces parallelism when targeting multicore platforms. It improves the programmability dividing application into several tasks. In addition, a thread is easier to create or destroy than a process. A simple representation of the thread life cycle is depicted in Figure 2.10.

Moreover, as a thread is a sub-entity of a process, it has a smaller context to save than the latter. Indeed, it does not have to manage global resources such as memory or CPU information. Thread context mainly includes registers and some other local values.

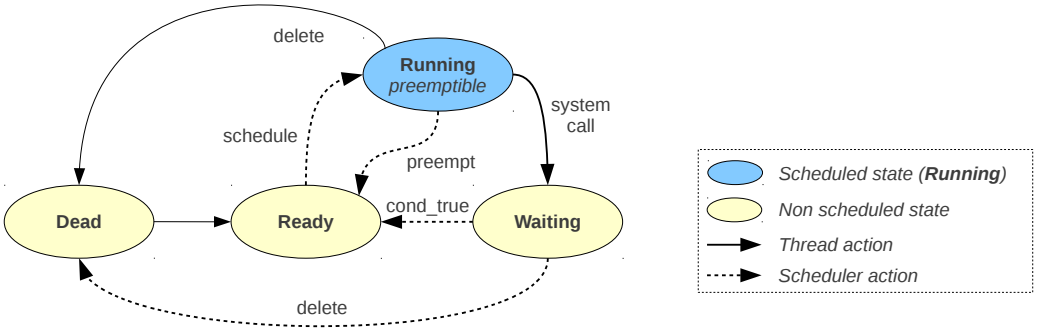


Figure 2.10: Thread life cycle

2.2.3 Software thread model

Generally, it exists two ways to implement a thread model in an operating system. Either in user space or in kernel space.

2.2.3.1 User thread model

In the user thread model, the operating system kernel is only aware of a single thread in the process. Threads are scheduled by a threads library implemented in the user space. The advantage of this model is that there is no need to modify the operating system, which is interesting if this one does not support the thread execution model.

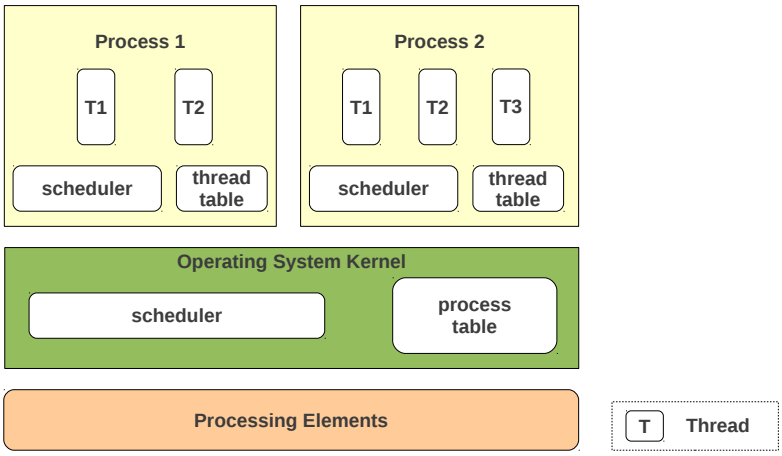


Figure 2.11: User Thread model

The user-level scheduler allows only one thread to be actively running in the process at a time. There is one thread table per process which allows a fast context switch as there is no need to request a kernel intervention *(Fig. 2.11)*. A local scheduling policy is possible but is limited. For instance, as the user-level scheduler

cannot manage a clock interrupt, a round-robin scheduling cannot be implemented. Regarding the parallelism, the main drawback of this model is that a blocking call from a thread would block all the threads implemented inside the same process.

2.2.3.2 Kernel thread model

In the kernel thread model, kernel threads are separated tasks which are associated with a process. In a kernel thread model, one kernel thread per process is created. The process table and the thread table are both managed at the kernel level. A preemptive scheduling policy is used in which the operating system decides which thread is eligible to share the processor.

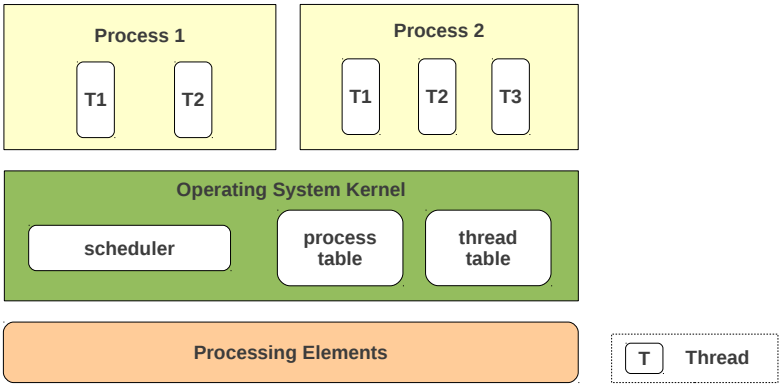


Figure 2.12: Kernel Thread model

Moreover, when a thread performs a blocking call, its state is notified to the kernel which can decide to preempt the thread in favor of another ready thread. As thread are managed at the kernel level, the drawback is that system calls costs are higher than in the user thread model.

2.2.3.3 Hybrid thread model

In a hybrid thread model, several user-level threads are running on top of a kernel thread *(Fig. 2.13)*. A commonly used hybrid thread model is the POSIX threads specification *(Pthreads)*. POSIX stands for Portable Operating System Interface. Threads are user-level threads but are managed using a kernel-assisted context-switching. It means that when a thread performs a system call, if the call is non-blocking, the thread rely on the user-level API. Otherwise, the kernel thread is notified that the thread is blocked and the kernel scheduler can try to find another process whose at least one thread is runnable. This solution is more complex to implement but tries to combine the best of the two models.

Finally, in embedded systems, the commonly used thread programming model is the kernel threads model. The goal is to reduce the memory footprint of the appli-

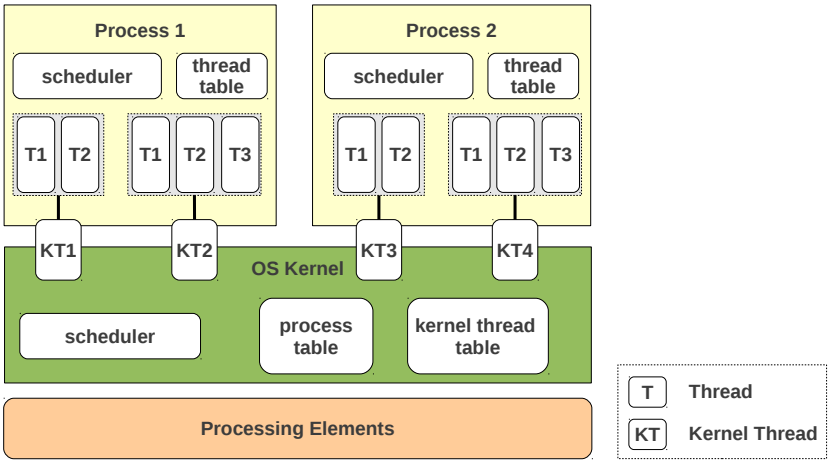


Figure 2.13: Hybrid Thread model

cation as the kernel thread structure is lighter than the process one. On the other hand, performance is lower due to the necessity to regularly switch from the user mode to the kernel mode. The hybrid thread model like POSIX tends to be adopted because the memory footprint become negligible regarding the available resources and above all because it is a widely used standard in the computing domain. The adoption of a standard being a good thing for the improvement of the applications portability.

2.2.4 Thread attributes

2.2.4.1 Storage structures

At the time of its creation, a thread is associated with two storage structures:

- a Data structure: Data is where all of the program variables are stored. It is broken down into storage for global and static variables *(static)*, storage for dynamically allocated storage *(heap)*, and storage for variables that are local to the function.

- a Stack structure: The stack contains data about the program or procedure call flow in a thread. The stack, along with local storage, is allocated for each thread created. While in use by a thread, the stack and local storage are considered to be thread resources. When the thread ends, these resources return to the process for subsequent use by another thread.

2.2.4.2 Thread-private data

Thread-private data are data that threads cannot share between themselves. Mainly, it includes the following resources:

- Thread identifier: A unique number that can be used to identify the thread.
- Priority: if the operating system allows specification of a thread priority, this value would determine the relative importance of one thread to other threads in the application.
- Call stack: The call stack contains data about the program flow or procedure call flow in the thread.

2.2.4.3 Thread-specific data *(TLS)*

Threads can have their own view of data items called thread-specific data. Thread-specific data is different from thread-private data. The threads implementation defines the thread-private data at the kernel level, while the application defines the thread-specific data. Threads do not share thread-specific storage, but all functions within that thread can access it.

Due to the design of the application, threads may not function correctly if they share the global storage of the application. If eliminating the global storage is not feasible, using thread-specific data is a good alternative.

2.2.5 Synchronization techniques among threads

Even if an application is thread-safe, in order to keep good performances, some global resources have to be shared between threads. In this case, the most important aspect of programming becomes the ability to synchronize threads. Synchronization is the cooperative act of two or more threads that ensures that each thread reaches a known point of operation regarding to other threads before continuing.

Threads can be synchronized using operating system services. These services ensure the developer that critical resources are accessed in a safe way and allow threads to communicate. The most common synchronization primitives are:

- Mutexes
- Semaphores
- Condition variables
- Threads as synchronization primitives
- Message Passing

2.2.5.1 Mutexes

A mutual exclusion *(mutex)* is a cooperative agreement between threads which ensures that only one of the threads is allowed to access the data or run certain

application code at a time. The mutex is usually logically associated with the data it protects by the application.

Create, lock, unlock, and delete are operations typically preformed on a mutex. Any thread that successfully locks the mutex is the owner until it unlocks the mutex. Any thread that attempts to lock the mutex waits until the owner unlocks the mutex. When the owner unlocks the mutex, control is returned to one waiting thread with that thread becoming the owner of the mutex. There can be only one owner of a mutex at a time.

2.2.5.2 Semaphores

Semaphores can be used to control access to shared resources. A semaphore can be thought of as an intelligent counter. Every semaphore has a current count, which is greater than or equal to zero.

Any thread can decrement the count locking or taking the semaphore. Attempting to decrement the count past 0 causes the thread that is calling to wait for another thread to unlock the semaphore. In the same way, any thread can increment the count unlocking or posting the semaphore. Posting a semaphore may wake up a waiting thread if there is one present.

In their simplest form *(with an initial count of 1)*, semaphores can be thought of as a mutual exclusion *(mutex)*. The important distinction between semaphores and mutexes is the concept of ownership. No ownership is associated with a semaphore. Unlike mutexes, it is possible for a thread that never took for the semaphore to post the semaphore.

2.2.5.3 Condition variables and threads

Condition variables allow threads to wait for certain events or conditions to occur and they notify other threads that are also waiting for the same events or conditions. The thread can wait on a condition variable and broadcast a condition such that one or all of the threads that are waiting on the condition variable become active.

Condition variables do not have ownership associated with them and are usually stateless. A stateless condition variable means that if a thread signals a condition variable to wake up a waiting thread when there currently are no waiting threads, the signal is discarded and no action is taken. The signal is effectively lost. It is possible for one thread to signal a condition immediately before a different thread begins waiting for it without any resulting action.

2.2.5.4 Threads as synchronization primitives

Threads themselves can be used as synchronization primitives when one thread specifically waits for another thread to complete. The waiting thread does not continue processing until the target thread has finished running all of its application code.

2.2.5.5 Message Passing

A message passing API can be implemented on top of the previous mechanisms. Threads can use this higher abstraction layer to synchronize and exchange data. This API provides blocking or non blocking primitives to transparently send or receive messages from a thread to another. Implementation can be realized using either the shared memory paradigm or a network protocol if a dedicated network is available.

2.2.6 Conclusion

Finally, to be considered as a software thread equivalent, the operating system managing the hardware threads has to provide them the ability to access to the same services than the software ones. The hardware thread model has to take it into account, and specifies additional mechanisms which allow the developer to process system calls.

2.3 Our Hardware Thread model

2.3.1 Context: The FOSFOR project

2.3.1.1 Presentation

The FOSFOR project is an ANR[7] project started in January 2008 and completed in December 2011. This is a collaboration between four partners: Thales Research and Technology France based in Palaiseau, the ETIS lab located in Cergy-Pontoise, the CAIRN from Lannion, and the LEAT based in Nice Sophia-Antipolis.

FOSFOR stands for Flexible Operating System FOr Reconfigurable platform. The aim of this project is to define a new kind of heterogeneous platform. This platform is heterogeneous in the sense that threads and operating systems could be implemented either in software *(running on one of the processors)*, or in hardware *(running in a partition of the FPGA)*.

Each part could then be adapted regarding the deployed application. The goal is to propose a homogeneous programming model for the application. This architecture is done to demonstrate the reconfigurable architecture viability regarding the development process complexity.

2.3.1.2 Platform architecture

The FOSFOR architecture is composed of multiple processing elements connected to a central bus *(Fig. 2.14)*. We distinguish software processing elements and hardware processing elements. Both implement respectfully a software and a hardware version of the RTEMS[8] [RTE 1988] operating system. On each processor, a software

[7] Agence Nationale pour la Recherche
[8] Real-Time Executive for Multiprocessor Systems

operating system manages classic software threads whereas a hardware operating system *(HwOS)* is able to manage reconfigurable partitions. Hardware accelerators are scheduled into these partitions.

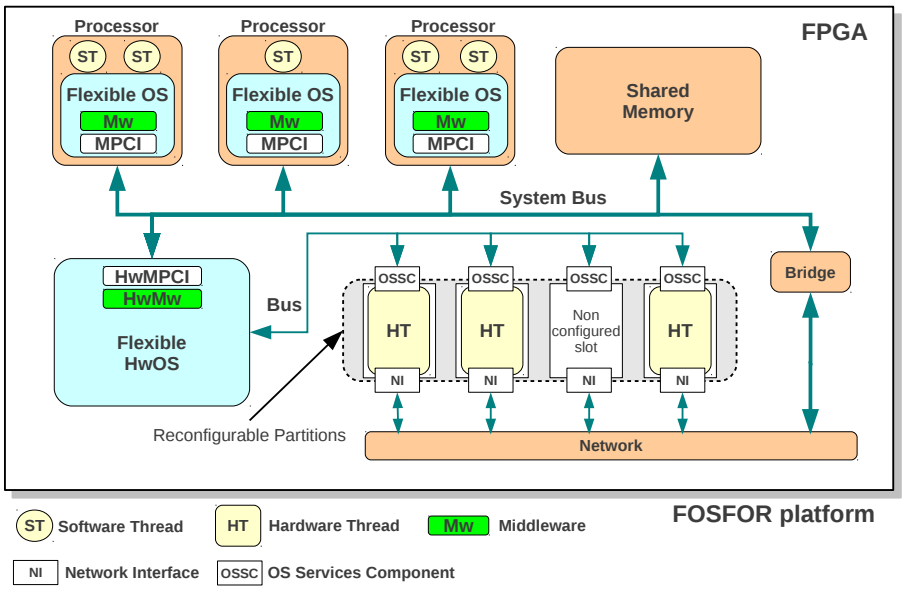


Figure 2.14: FOSFOR platform architecture

The objective is to provide at the user-level a homogeneous thread point of view. To achieve it, we abstract hardware accelerators into hardware threads. The architecture of these hardware threads is defined in details in Sections 2.3.2 and 2.3.3.

2.3.1.3 High-level communication mechanisms

Communication between threads can be handled using two ways. For synchronization and small data transfer, threads can rely on the operating system services. These services can be locally managed or shared between all processing elements. For larger amount of data, a middleware layer provides a message passing API with Send and Receive primitives.

This middleware layer *(Mw)* is inserted between the application layer, based on POSIX threads, and the operating system services API. If a thread wants to communicate with another one, it has access to the simple middleware API using transparent message passing protocol, or it can access directly to the operating system services, such as Mutex or Message Queues primitives.

This high-level API composed of these two types of primitives has been ported on the hardware side. From the user point of view, the application is only composed of threads. Starting from here, an automatic tool can be expected to generate both

software and hardware code. For instance, basing the description of the application on the components can be a good solution to facilitate the implementation of heterogeneous applications on HRSoC platforms.

In software, the MPCI[9] layer included in RTEMS is the base of the heterogeneous communication. It provides a transparent access to distant services. We extended it to the hardware implementation of the services. The bridge has to be transparent to abstract both the location and the heterogeneity of the application threads. The location of each hardware thread which dynamically changes regarding the available slots is dynamically managed and abstracted by the middleware layer.

2.3.2 Hardware Thread specifications

2.3.2.1 Objectives

In order to simplify the programming complexity of the HRSoC, hardware accelerators have to adopt the same behaviour as their software counterparts. To do so, they should be able to obey the orders of the operating system. They also must have the ability to call operating system services available in the whole platform, read and write data from and to memories, and specifically they should be associated with an interface allowing the developer to control the execution of these hardware accelerators. The hardware thread life cycle should be equivalent to the software one. All these features and interfaces are assembled in order to encapsulate the accelerator and so to define what we call a hardware thread.

2.3.2.2 Definition

We defined a hardware thread to take advantage of the dynamic reconfiguration provided, for instance, in the Xilinx FPGAs. It is composed of two main parts: a static part which contains all the interfaces with the platform, and a dynamic application-specific part, which contains the Accelerator, the Finite State Machine *(FSM)* controlling its execution, and a private memory *(Fig. 2.15)*.

Compared to a software thread, a hardware thread will run on a reconfigurable partition. This reconfigurable partition can be compared to a process, in which the logic resources are equals to the processor resources shared between every threads running inside this process. In this scheme, a set of reconfigurable partitions is a processing element containing several processor cores. A parallel can be done between a reconfigurable partition and a processor core.

Static interfaces correspond to the user-level API. It provides to the thread an access to the operating system and communication services. The User FSM is the sequential code executed by the thread and finally, the double port memory connected both to the Accelerator and the Network Interface is used as the heap and stack storage by the thread.

[9] Multi-Processor Communication Interface

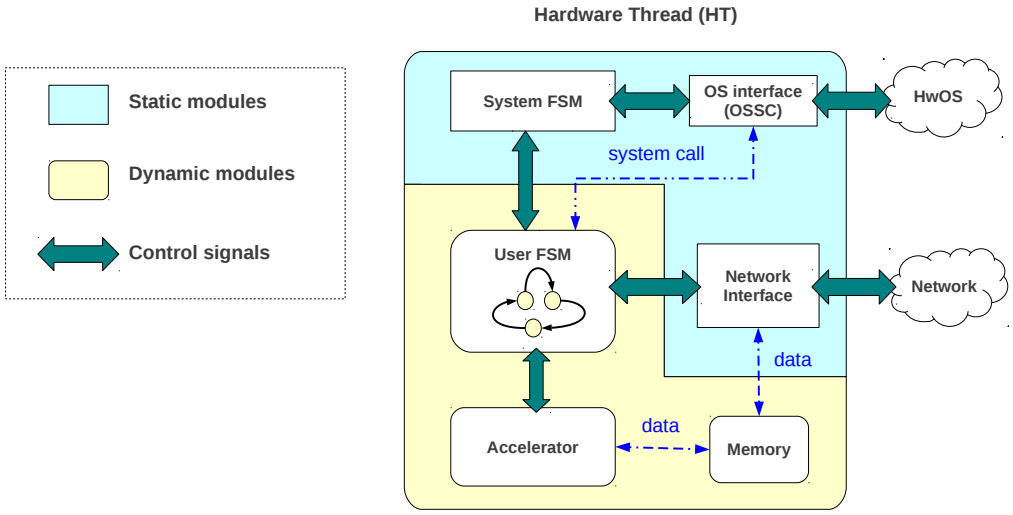


Figure 2.15: Hardware Thread Architecture

2.3.3 Hardware Thread architecture

2.3.3.1 Operating System interface

The static part provides an interface to interact with the operating system *(OS Interface)*. It contains a specific component responsible for implementing the protocol between a thread and the operating system called the OS Services Component *(OSSC) (Fig. 2.16)*. This interface is the same for all threads and considered as static. In this way, it can be reusable by any hardware thread and so it can ease the thread preemption process. It is composed of a standard dual-port memory in which the thread can write the identifier of a system call and its parameters. Upon notification via a "SysCall" wire, call parameters are read by the operating system using a dedicated bus that connects all the reconfigurable partitions. Once the system call is done, return values are written back in the memory and are read by the thread.

This protocol allows a hardware module to perform system calls with the exact same semantic as pure software. Therefore we can implement a consistent API for both hardware and software threads, and greatly reduce the heterogeneity gap.

2.3.3.2 System FSM

Once instantiated by the operating system, this one can control the System FSM of the hardware thread to handle the accelerator execution. This FSM supports four basic commands: start, suspend, resume and stop. Dynamic and partial reconfiguration feature provided by Xilinx FPGAs allows the scheduler to dynamically reconfigure the dynamic part of the thread and so to temporarily share a given partition between several threads, as it is done on the software side where a CPU is

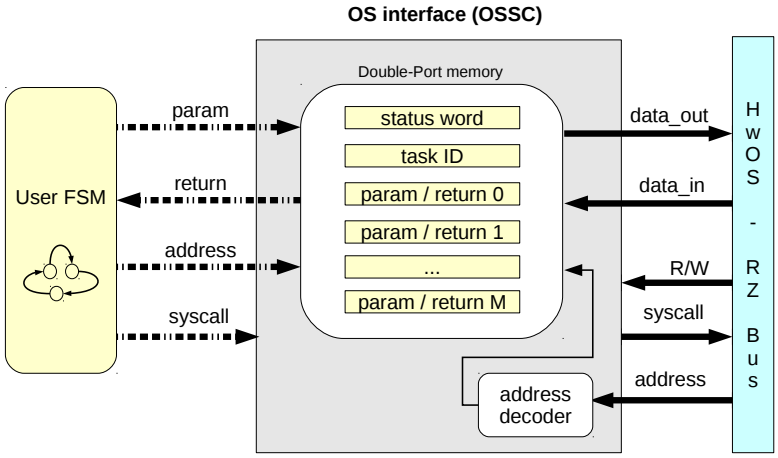


Figure 2.16: OSSC architecture

shared by software threads.

2.3.3.3 Hardware Thread life cycle

However, a hardware thread has a specific life cycle compared to a software thread. The operating system has to be able to manage temporarily and spatially the hardware thread, so it has to take into account if the thread is running and if it is configured or not. Nonetheless, orders given by the operating system to a hardware thread are the same as the ones given to a software one and include starting, stopping, or resuming its activity.

From the scheduler point of view, a hardware thread has the same three classical states than a software thread: Ready, Waiting and Running. But as the components of a hardware thread are inherently parallel, and it is located in a dynamically reconfigurable partition, we added new states to take into account the configuration status, as shown in *Fig. 2.17*.

In order to mitigate the reconfiguration latency, the scheduler can choose to keep a thread configured while it is waiting on a blocking system call. Furthermore, in order to keep the network complexity at a manageable level, we forbid the preemption of a hardware thread while there is a pending communication with this thread. This is why we refine the Waiting state into three states: configured and non-preemptible, configured and preemptible, and non configured. The ability for a thread to claim itself as non-preemptible when communicating involves to limit the size of the packets exchanged on the network, this in order to ensure that a thread cannot monopolize a reconfigurable slot.

Similarly, the Running state is refined into preemptible and non-preemptible states. To simplify the scheduling management, a hardware thread comes back into

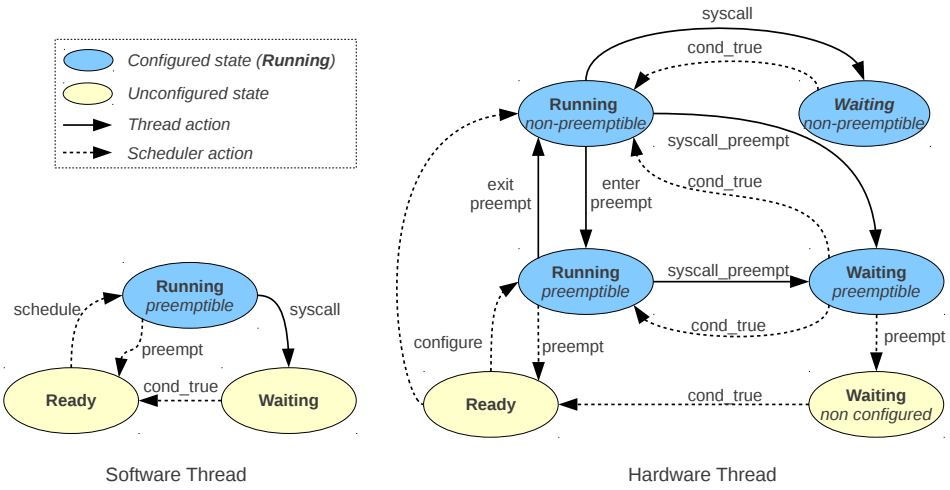


Figure 2.17: Software and Hardware Thread States

a preemptible state by notifying it explicitly with a blocking system call or a specific primitive.

2.3.3.4 User FSM

The user FSM defines the behaviour of the hardware thread. States defined into this FSM allow the user to process system calls, send or receive data to or from the network and order the accelerator to perform its function.

It can be controlled by an external operating system via a control register. The operating system is therefore able to start, restart, suspend or resume the thread execution. This control register is mapped in the OSSC memory at offset 0x00 *(Fig. 2.16)*.

In Figure 2.18, blue ellipses correspond to the states which are presents in the System FSM of the thread. The *RUNNING* state allows the Processing Logic Element execution, *RECV* and *SEND* states are middleware calls, whereas *LOCK* and *UNLOCK* are classical mutex primitives. In order to facilitate task implementation for the developer, we provide a VHDL package including every available system calls procedures. These procedures are responsible of writing call parameters in the OSSC memory and to get back return values. This set of procedures, coupled with the common interfaces, allows to easily generate the hardware thread source code *(Fig. 2.19)*.

2.3.3.5 Network interface

A hardware thread would be commonly used in order to process large amount of data. To be useful, it should provide an efficient way of communication. A network

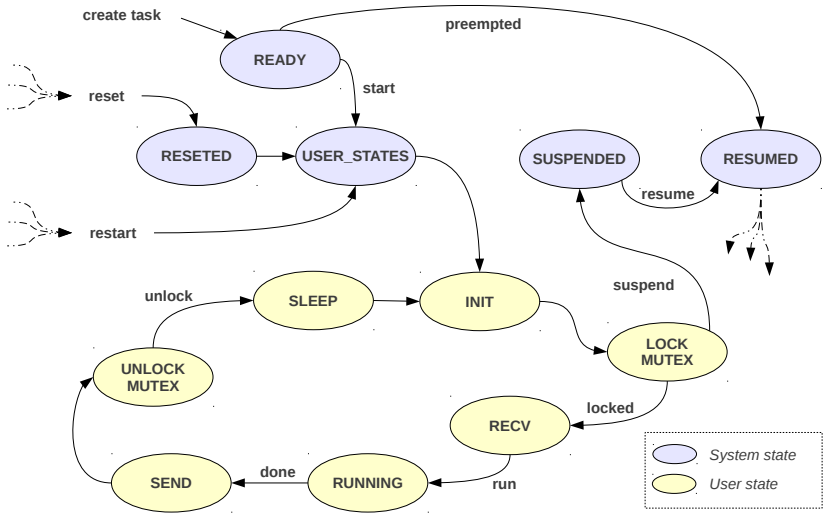


Figure 2.18: Hardware Thread FSM example

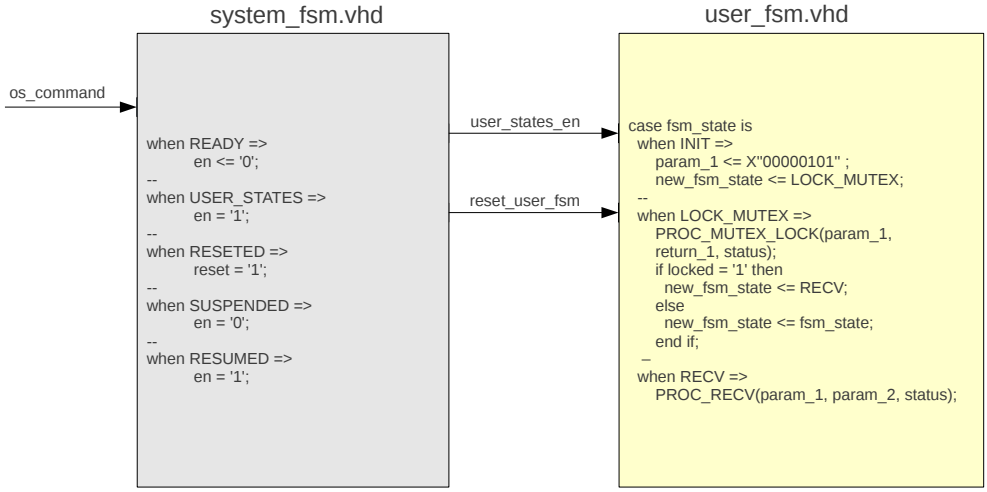


Figure 2.19: Hardware Thread HDL files example

interface is responsible for creating and decoding packets, sent and received, to and from a dedicated network. Because threads inside a HRSoC can run in parallel, the ideal environment for a hardware thread is a networked system in which it could communicate with another hardware thread from point to point.

The Network Interface *(NI)* is the static interface of the Hardware Thread *(Fig. 2.15)*. It is connected to a dedicated Network-on-Chip *(NoC)*[Devaux 2009] implemented in order to offer a fast medium of communication between hardware threads and memories. It also ensures them a fast way of communication with software threads.

NI architecture is shown in *Fig. 2.20*. Two FIFOs allow the User FSM to stack Send and Receive requests. These requests are then respectively processed by a Packetizer and a Depacketizer. A DMA is connected to one of the port of the thread internal memory. This DMA is driven on one hand by the Packetizer to read data in memory and send it through the NoC and on the other hand by the Depacketizer to receive data from the NoC and writes it into the internal memory.

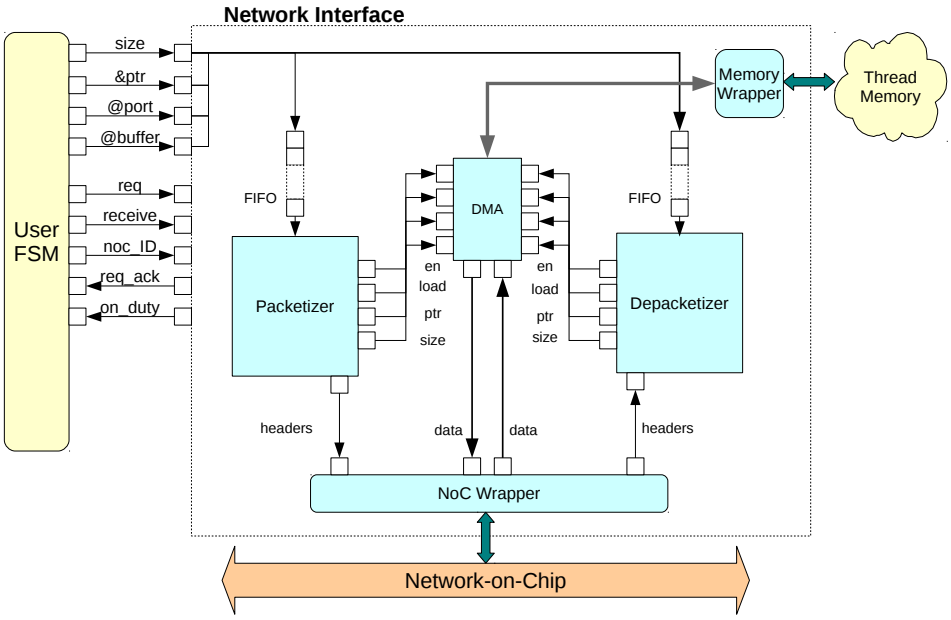


Figure 2.20: Network Interface architecture

Elements connected to the NoC communicate through it by sending data packets over the network. We specified and implemented a protocol to provide two main features to a thread. The first one consists in sending data to an element connected to the NoC. This element can be another thread, a memory or another kind of peripheral. The second is receiving data from another element. In the case of a passive element, such as a memory connected on a upper leaf of the NoC, we defined special packets to allow write operation in one shot and read operation in two phases : one request from the thread and an answer from the memory. The protocol adopted for the network has been thought to provide these two features in a transparent way for the developer. This protocol is detailed in Appendix A.1.

2.4 Hardware Thread programming model

2.4.1 Operating System services protocol

2.4.1.1 Thread commands

When the operating system wants to send an order to a hardware thread, it has to write it in the OS Services Component *(OSSC)* memory *(Fig. 2.16)*. The first word of this memory is the Status Word. The Status Word entry is both used to receive the operating system commands and the system call status code. The content of the latter is explained in Section 2.4.1.2. The content of the former is detailed in Figure 2.21. The operating system command is always given priority over the thread system call. In this case, the command field in the status word is decoded by the OSSC and the order is transmitted to the System FSM which controls the state of the User FSM. Just after its configuration, the hardware thread is set in a ready state by the System FSM.

Bits	31 - 8	7 - 5	4 - 0
Content	(ignored)	command	status code

Figure 2.21: OSSC Status Word content

When receiving the Start command, the thread goes to the INIT state which must be defined by the developer *(Fig. 2.18)*. When receiving the Suspend command, the hardware is stuck in its current state by the System FSM and no registers modification is allowed in this state. This command can be used to save the context of the thread.

2.4.1.2 User API

On the other side of the OSSC, when a thread wants to process a system call, it must call a VHDL procedure which will transparently handle the OS interface. A procedure takes as parameters the primitive parameters, the registers to store return values as well as control signals to communicate with the OSSC *(Fig. 2.22)*.

A procedure is a Finite State Machine which stacks the parameters and the requested service in the OSSC memory *(Fig. 2.23)*. Then it waits for the operating system acknowledge. Actually, this acknowledge is the detection by the OSSC of a read request from the operating system. After this, the procedure waits for the operating system to answer. Even if the thread processes a non-blocking call, it will receive a Status Code notifying if the call succeeded or not. This Status Code is defined by the underlying operating system. It could be a value like OS_SUCCESSFUL, OS_FAILURE or something specific to the requested service

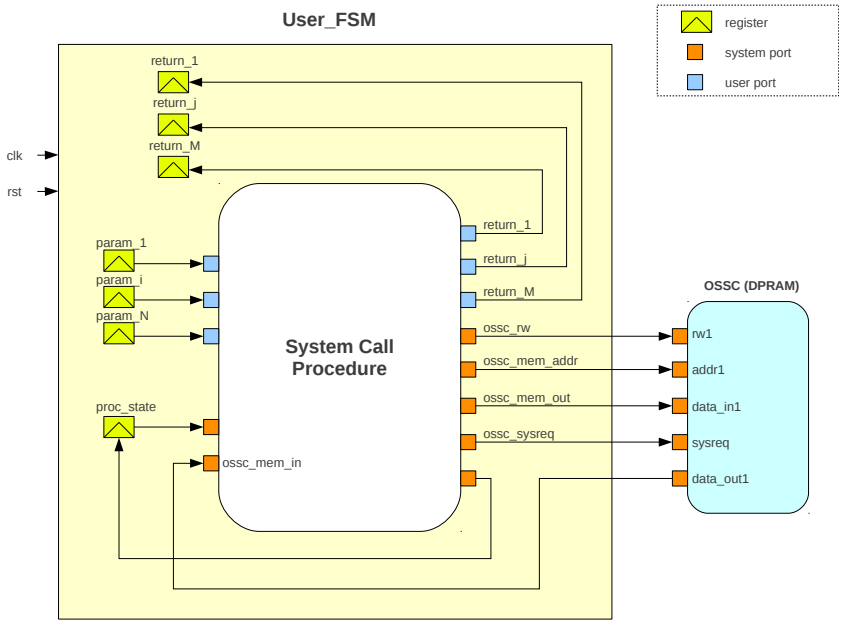


Figure 2.22: System Call procedure

like ERR_MUTEX_INVALID or ERR_MSG_QUEUE_FULL.

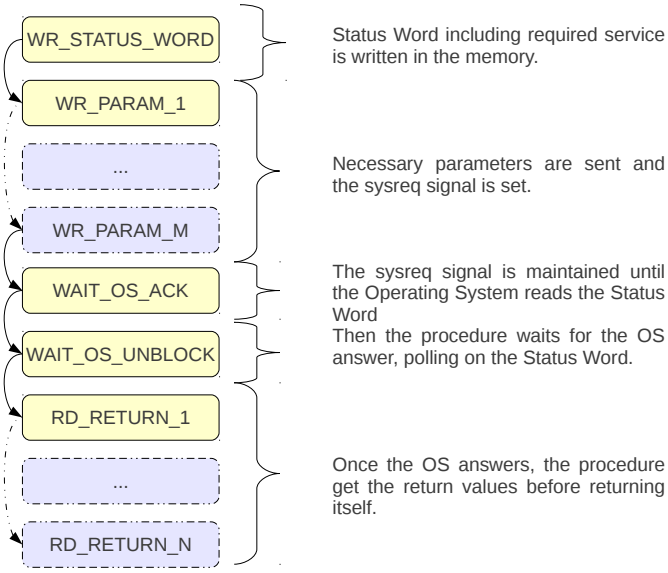


Figure 2.23: System Call procedure steps

The advantage when dissociating the user code from the operating system specificity is that the application can be easily ported on another operating system or API. It just requires some modification in the OS Services Component to be com-

pliant with the targeted operating system.

2.4.2 Network communication protocol

In the context of the FOSFOR project, we had to define the structure and the behaviour of the interface between a hardware thread and DRAFT Network-on-Chip *(NoC)*. This interface should allow threads to process non-blocking send and receive requests. All requests are processed sequentially without interruption. Namely, for a given request, all packets are sent or received the one after the other.

The maximum packet size is defined at configuration time, namely when synthesizing the hardware platform. The Network Interface can access to the internal memory of the thread. This access allows it to load data from the memory in order to send it over the network and to get data from the network in order to store it in the internal memory.

A common abstraction level has been defined in the middleware layer. The communication processes defined in software are adapted in hardware to allow them to communicate in a transparent way with the software threads.

2.4.2.1 Supported requests

The two first basic requests supported by the Network Interface are the Send and Receive primitives *(Fig. 2.24)*. A Send request consists in sending one or several packets over the network. Packet size is fixed by communication medium design *(ie. the NoC)*. A packet is composed of a header followed by data to transmit. Data and header are represented by 32-bit width flits. A receive request consists in waiting for a packet to come from the network. It is a passive request which involves no transmission from the requesting thread.

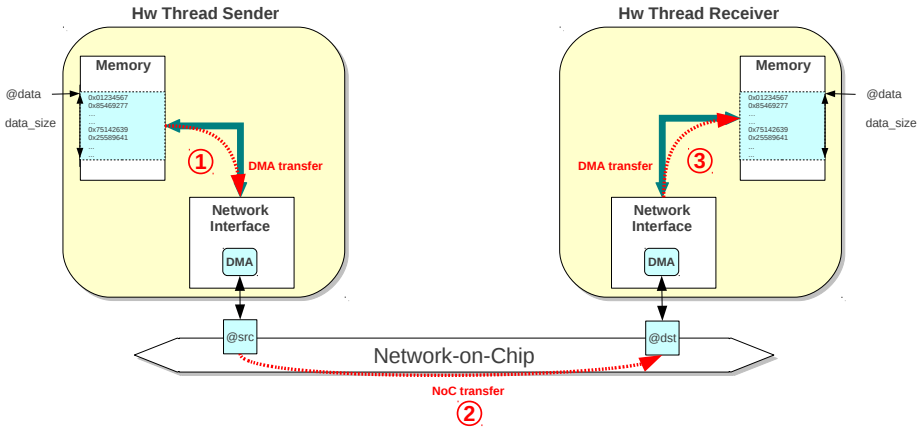


Figure 2.24: Network Interface Send and Receive protocol

The two others supported requests are the Write and Read primitives *(Fig.*

2.25). A Write request is similar to a Send request except that additional header flits are sent after the two main flits. The main flits are essential to ensure a correct routing inside the network. The first flit contains the sender and receiver port address whereas the second one contains the number of flits included in the packet. Additional headers indicates the address of the targeted buffer in the remote memory and the number of words to read or write.

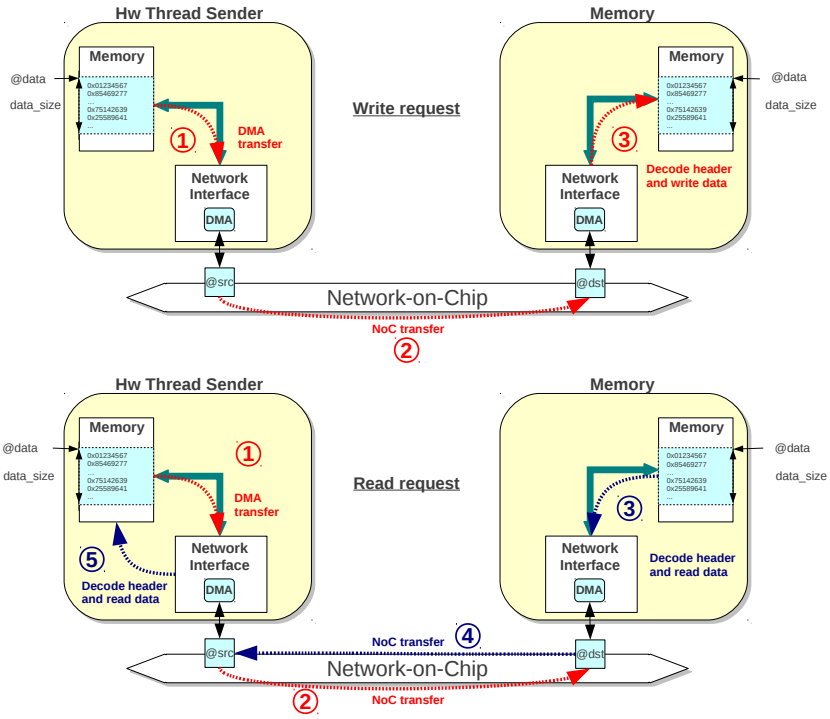


Figure 2.25: Network Interface Write and Read protocol

This protocol allows a hardware thread to efficiently and directly communicate with others hardware threads and also to exchange data with software threads using memory buffers. The bridge visible in Figure 2.14 is responsible for translating the messages between the processor bus and the network-on-chip. It permits to get a homogeneous API and memory mapping between the software and hardware threads.

2.4.3 Accelerator interface

The accelerator represents the main function of the hardware thread. As the internal structure of a hardware thread is inherently parallel, we implement pipeline mechanisms between this function and the communication features in order to take advantage of this parallelism *(Fig. 2.26)*.

In the hardware thread, user application is described in the User FSM. This is

in this state machine that the user is allowed to process operating system calls and to be able to perform pipelining. From a functional point of view, as the network-on-chip is full-duplex, the user is able to simultaneously run the Network Interface to send or receive data, and the hardware accelerator to process these data taking advantage of the double port internal memory.

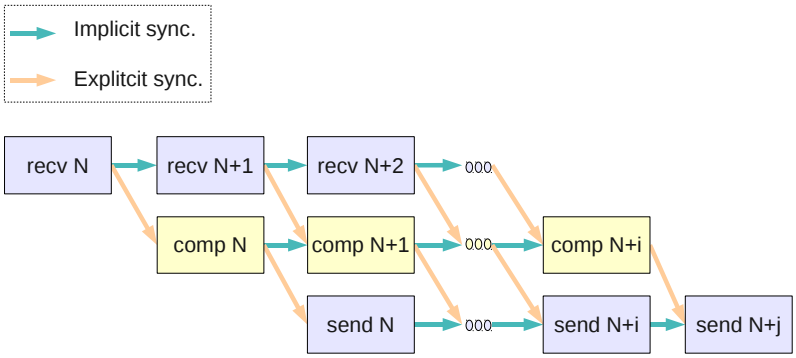


Figure 2.26: Parallel processing using pipelining

To do so, the three modules which are the Network Interface Packetizer, the Depacketizer and the Processing Logic Element, should be able to synchronize. Each module is connected to a synchronization controller *(Fig. 2.27)*. This controller has a FIFO in which the User FSM can stack execution requests. The controller process each request one after the other, which will implicitly synchronize execution requests for a given module.

On the other hand, synchronization between two modules is explicitly expressed in the user request. This request should contain two pieces of information:

- conditions to start a request processing, it means with which module a synchronization is necessary and how many synchronization have to be realized.

- and a mask which identify modules waiting for a synchronization in order to notify them the end of the process.

At the User FSM level, when the user wants to send or receive data, as well as process it, he just have to send a simple request to the synchronization controller. Once the request is buffered, the controller sends back an acknowledge which permits to the User FSM to send a new request or to call an operating service without waiting the end of the request processing.

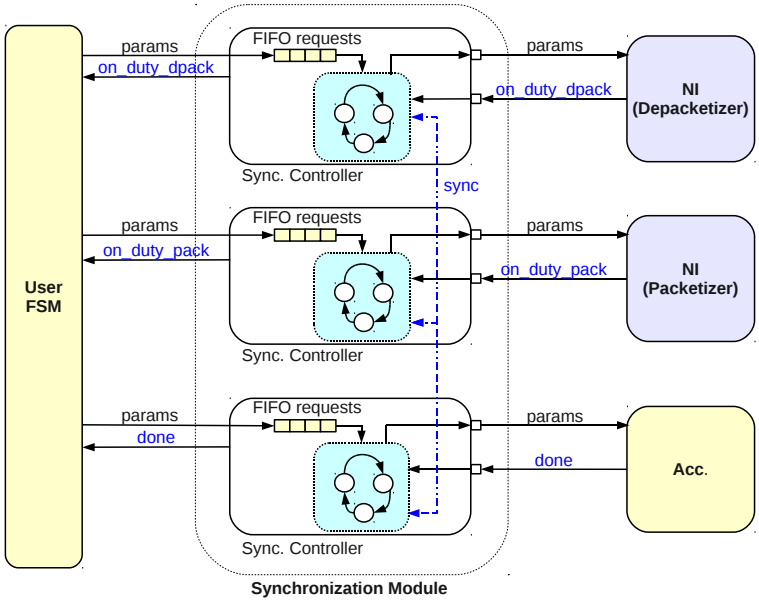


Figure 2.27: Synchronization Module

2.5 Conclusion

In this chapter we proposed a thread model which can access to operating system services and whose the behaviour is close to a software one. Moreover, we introduced an additional programming model inspired from the Synchronous Data-Flow model. This allows to program hardware threads with a higher level programming model than the thread model.

All these features allow us to consider a hardware accelerator as a real thread. Obviously, even if they are similar, at low-level software and hardware threads cannot be managed in the same way by an operating system. This is why we need dedicated techniques to manipulate hardware threads, especially when the objective is to be able to preempt them on the available slots provided by the platform. This issue is the topic of the next chapter.

Chapter 3

Hardware threads preemption using Dynamic and Partial Reconfiguration

Contents

3.1 Introduction

Numerous works have been done about the management of the dynamic and partial reconfiguration, especially the preemption of hardware accelerators. These works rely on the definition of hardware accelerators as hardware threads, and define

mechanisms to provide this important feature to handle these threads on top of an operating system.

In parallel, design tool flows had to be rethought in order to offer the ability to design reconfigurable platforms composed of independent reconfigurable slots. This low-level feature would ease the hardware threads scheduling by an operating system providing a flexible way to relocate module from one slot to another.

These two points are discussed in this chapter. A state-of-the-art of different preemption mechanisms is introduced in Section 3.2. Section 3.4 details our work targeting the Virtex 5 FPGA family. Then the design flow issue is tackled in Section 3.5 where problematic and solutions are detailed before to be implemented. Finally, Section 3.6 summarizes our contribution and explains how it can be integrated into an operating system architecture, making a link to the next chapter.

3.2 Related works

3.2.1 Preemption mechanisms

In [Kühnle 2006], Becker et al. include in their dynamically reconfigurable system described in [Grimm 2004] a 2-Dimensional *(2D)* module relocator. In this system a module can only be relocated on an area whose resources are horizontally homogeneous regarding its point of origin. As they target Xilinx devices, FPGA resources are always vertically homogeneous.

In particular they target the Virtex-2 Pro FPGA family. However, one of the specificity of these chips is that, on contrary of latest FPGAs, they can only be dynamically reconfigured in 1-Dimensional *(1D)*, that is to say it is necessary to reconfigure a whole column of resources. To overcome this issue and to offer 2D relocation mechanisms, they propose to use the read-modify and write-back method. It consists in reading back the module configuration, modifying it to place the module in the wanted area and writing-back the bitstream.

The glitch-less property of the reconfigurable matrix ensures that if the same data than the one which is currently configured is written in the memory, no glitch would disturb this data. Due to this property, they can modify one module of a column without changing the others implemented in the same column.

The bitstream is modified using the Jbits software [Guccione 1999] provided by Xilinx for Virtex-II devices which is running in parallel on a computer. This bitstream is sent through the UART. From here, the downloaded bitstream does not contain information about the bitstream location. It is added later by the relocator.

Another issue raised in this paper is the flexibility needed for the connection between the dynamic module and the rest of the system. To leverage it, Virtual Routing Channels are placed near each module. When reconfiguring the system, the router near the module can be dynamically linked to one of the slots of the routing channel *(Fig. 3.1)*. These slots can be reconfigured according to modules needs and

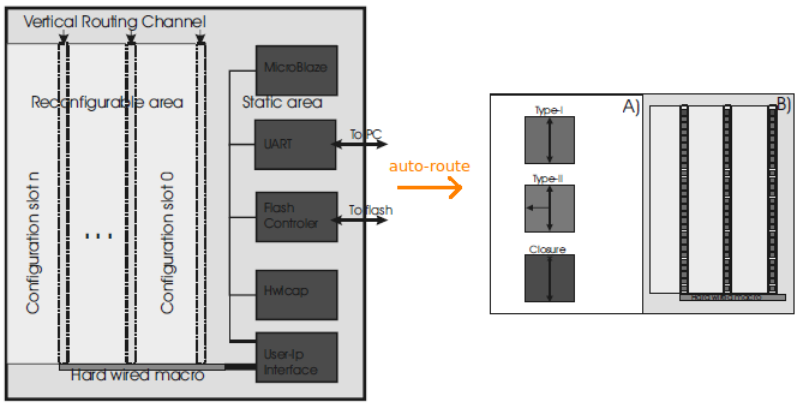


Figure 3.1: Virtual Routing Channels

so allow to connect several modules of the same column.

The paper [S. Corbetta M. Morandi M. Novati 2009] also deals with the 2D bitstream relocation. The introduced relocator is named *Birf* and has been realized to be used on Virtex 4 and 5. The relocation is limited to Slices-based modules. The use of BRAMs and DSPs in reconfigurable modules is prohibited in this design flow. There are no specific design constraints as this flow relies on old Xilinx tools options *(10.1 and above)*, which unfortunately are not available in current tools anymore *(13.1 and up)*.

In [T. Becker 2007], authors perform task relocation on strictly homogeneous areas. The relocator comes in two versions : a software one and a hardware one. If it exists non homogeneous resources from one partition to another, it means BRAMs or DSPs, relocation can still be done if they are not used by the design. In this case, columns are considered as empty and not reconfigured by the relocator.

In [Kallam 2009], the authors propose a method to relocate dynamic modules on the fly. The principle is to perform a read-back of the bitstream, word after word *(a word is 32 bits wide)*. Then the relocator processes each word in a pipeline in order to relocate it in the destination area. It offers a gain in memory occupancy as there is no need to store the totality of the partial bitstream anymore.

This is also interesting regarding the execution speed because using a BRAM as storage memory offers a very fast access to data. This technique is well suited for module relocation but does not allow to implement preemption mechanisms. However it has the advantage to show that dynamic reconfiguration speed can be improved regarding the original ICAP controller provided by Xilinx. This controller can access to high-capacity memory storage but is processor-dependent. Accelerating the partial reconfiguration process would allow to target a larger range of real-time application.

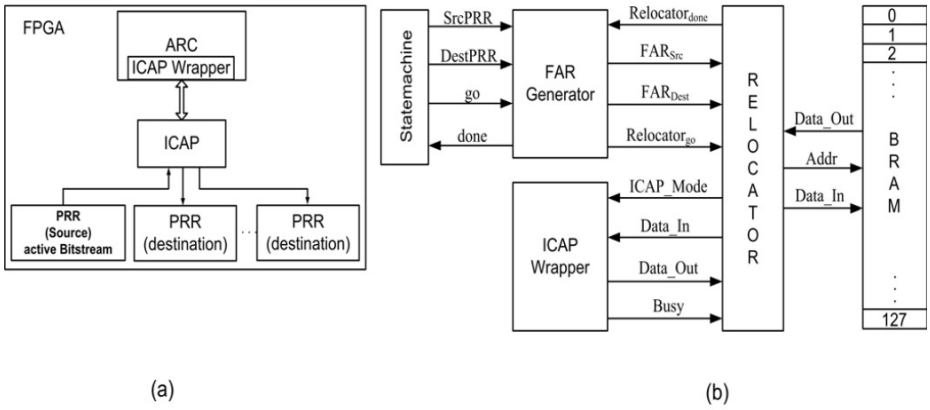


Figure 3.2: (a) Implementation of PRR-PRR relocation (b) Top-Level block diagram of ARC [Kallam 2009]

3.2.2 Reconfiguration accelerators

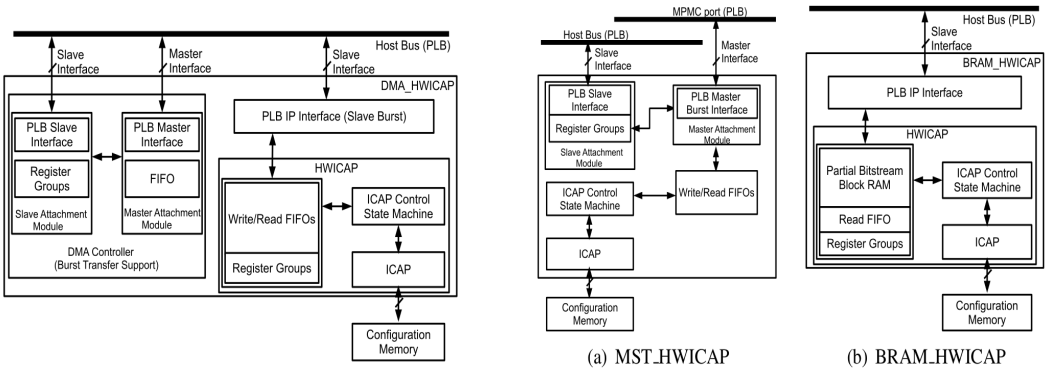
[Liu 2009] proposes a design space exploration of different architectures to improve the configuration time of the ICAP driver when loading a partial bitstream. All tests have been done on a Virtex 4 platform *(ML405 Development Board)*. In addition to the IP cores provided by Xilinx, the OPB HwICAP and the XPS HwICAP, three other solutions are asserted *(Fig. 3.3)*.

The first solution is based on a DMA[1] engine running aside of the HwICAP controller and feeding it through the processor bus. The second one is a customized instance of the XPS HwICAP which allows it to be master on the processor bus and so act like a DMA controller. Finally, the last architecture consists in adding a Block RAM inside the ICAP controller in order to store the bitstream and so dramatically reduce the transfer latency.

In summary, the DMA and the Master architectures respectively offer a speed-up of 5.5 and 16 regarding a Microblaze-platform using cache memory and the XPS HwICAP. As for the BRAM solution, it permits to reach the theoretical limits of the reconfiguration port but it must be reserved for small reconfigurable modules.

FaRM [Duhem 2011] stands for Fast Reconfiguration Manager. This component is master on the PLB bus and so can directly access to an external memory *(Fig. 3.4)*. FIFOs are implemented to store the partial bitstream and allow to process pre-fetch load to hide a part of the reconfiguration overhead. It relies on compression without loss techniques to reduce the amount of data to store in the FIFO. Moreover, even if Xilinx recommends to operate the ICAP at 100MHz and so allows

[1] Direct Memory Access



(a) MST_HWICAP (b) BRAM_HWICAP

Figure 3.3: ICAP accelerators solutions [Liu 2009]

a maximum throughput of 400MB/s, the solution provides by FaRM can overclock the ICAP at 200 MHz and so allows in certain cases to process configuration with a maximum throughput of 800 MB/s.

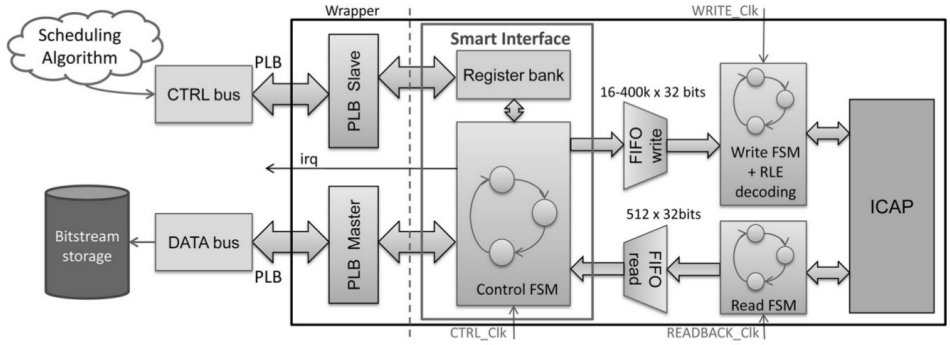


Figure 3.4: FaRM architecture [Duhem 2011]

Uparc [Bonamy 2012] stands for Ultra-Fast Power-aware Reconfiguration Controller. Like FaRM this controller relies on decompression techniques and overclocking to enhance the reconfiguration process. However, they use a better compression algorithm and a customized BRAM for bitstream storage *(Fig. 3.5)*. The latter permits them to overclock the ICAP at 362.5 MHz and so to reach a maximum throughput of 1433 MB/s without compression, and 1008 MB/s with compression. In the last case, the BRAM size of 256 KB allows to store maximum bitstream of 992 KB which is more than 40% of the Virtex 5 SX50T FPGA full configuration file.

The solution offered by Koch et al. [Hansen 2011] is currently the most efficient regarding the reconfiguration throughput. Their accelerator provides a wider data path size than the original ICAP primitive extending it to 64 bits. In addition they also rely on two different clocks: the first one to fetch the input data, and the sec-

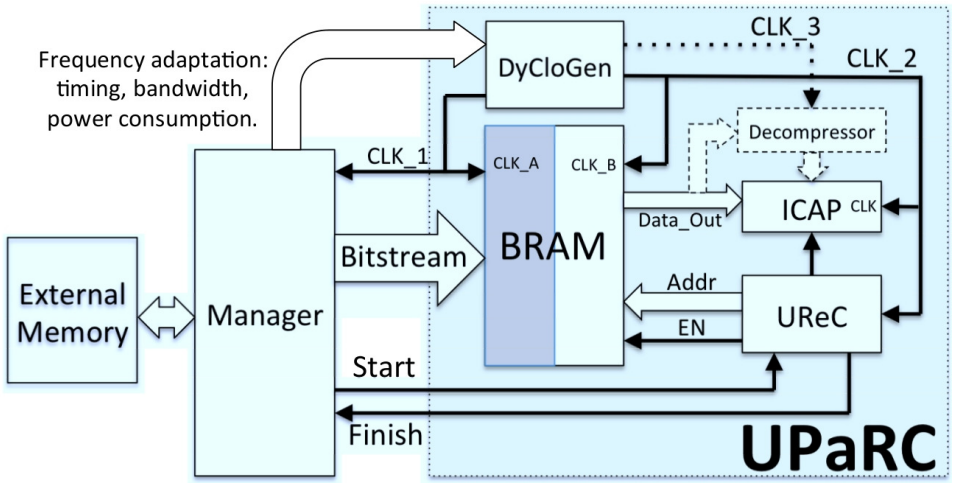


Figure 3.5: Uparc architecture [Bonamy 2012]

ond one which must operate at twice the frequency of that of the first one *(Fig. 3.6)*.

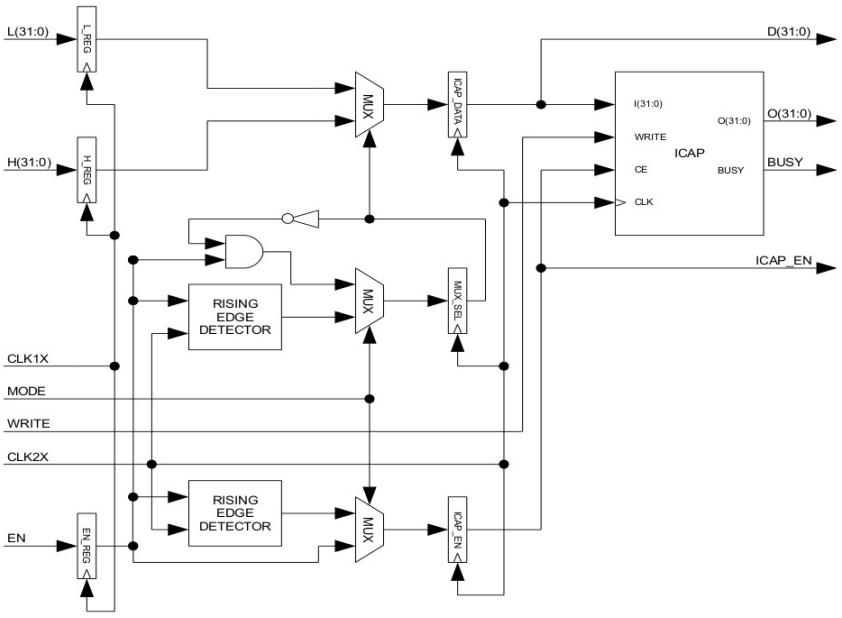


Figure 3.6: ICAP Hard Macro block diagram [Hansen 2011]

On a Virtex 5 platform, they create a Hard Macro and achieve to feed the ICAP with a clock running at 550 MHz. Above this frequency, the configuration process freezes and it is necessary to reboot the ICAP. This architecture allows them to reach a throughput of 2200 MB/s. This throughput can only be achieved with bitstream whose the size can be contained into the FIFO of 64 KB. However, in future work the

addition of DMA mechanisms and decompression engine could lead to improve the access to bigger external storage memories. Also, it should be noticed that during readback, in order to ensure that the reconfiguration memory of the FPGA is read correctly, this operation is still done at 100 MHz, that is to say at the classical throughput.

3.2.3 Design tools

Regarding the design tools, the standard ones provided by Xilinx do not offer the ability to design independent and relocatable dynamic modules. Accordingly, new tools have been developed by the community to permit the development of design using alternative constraints necessary to control at fine-grain the placement and the routing of both the static and the dynamic modules.

RapidSmith [Lavin 2011] is a JAVA API which allows to manipulate XDL[2] files. It relies on a complete database about the resources architectures of each Xilinx FPGA devices *(Fig. 3.7)*. It allows the developer to implement its own placer-router going from the XDL format to the proprietary NCD[3] format used by the Xilinx tools.

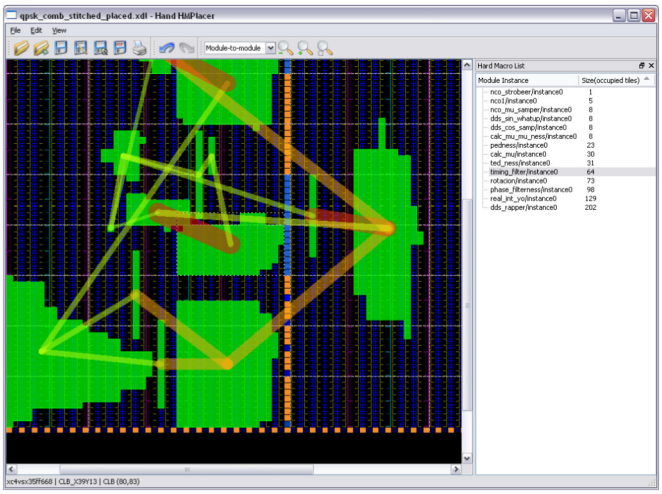


Figure 3.7: RapidSmith screen capture [Lavin 2011]

OpenPR [Sohanghpurwala 2011] is an open-source software based on the same routing engine which permits to create independent partition using blocker macros [Koch 2010a]. These macros prevent the static routing from crossing inside the reconfigurable partitions *(Fig. 3.8)*. However this tool is not integrated into the standard flow and the number of supported devices is limited.

[2] Xilinx Description Language
[3] Native Circuit Description

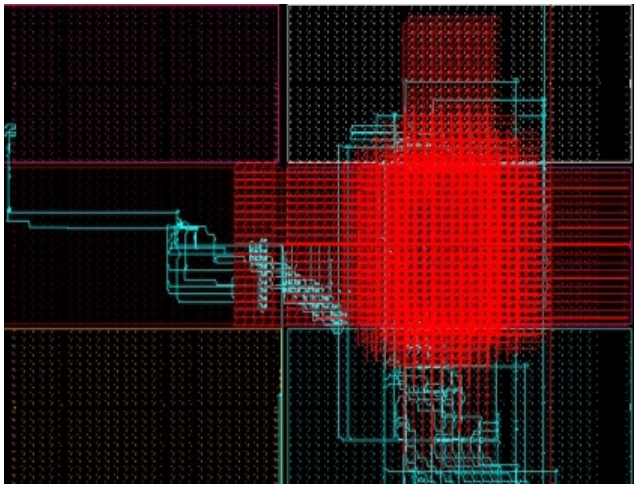


Figure 3.8: OpenPR screen capture from FPGA Editor [Sohanghpurwala 2011]

In order to keep using the Xilinx tools and the support of all existing devices, the new Isolation Design Flow [Corbett 2012] provides additional options to segregate the partitions in a design. Segregation includes both the use of the logical resources and the use of the routing resources *(Fig. 3.9)*. Basically, it was created to tackle security issues in cryptographic systems, allowing safe function duplication. However, it can be deflected in order to design homogeneous and relocatable partitions.

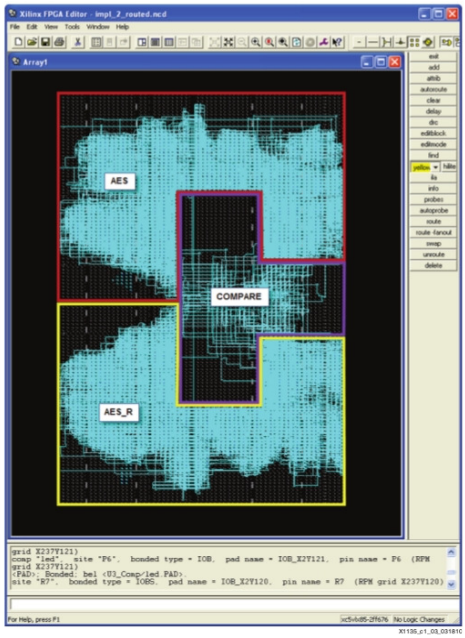


Figure 3.9: Isolation Design Flow screen capture from FPGA Editor [Corbett 2012]

All these design tools have been experimented to process module relocation in Virtex 5 FPGA devices. Results and comparisons are detailed in Section 3.5.4.

3.3 FPGA reconfiguration knowledge

Our objective is to provide preemption mechanisms on Virtex 5 FPGA, which could be applied in the future to Virtex 6 or even Virtex 7 FPGA with minimal changes in the procedure. To do so, we first have to be well aware of how this kind of FPGA is reconfigured and particularly how the configuration bitstream is organized. This is the topic of the next section which will then lead us to propose preemption mechanisms, before tackling the design flow issue.

3.3.1 Virtex 5 FPGA resources

Logic resources are mainly represented by the Configuration Logic Block *(CLB)*. A CLB is a processing unit of Xilinx FPGAs. Each CLB is connected to a global switch matrix for signal routing and is composed of two Slices. It exists two types of Slices *(Fig. 3.10)*: whereas Slices-L are slices containing logic elements, namely LUTs[4] and Flip-Flops, Slices-M are slices done to be used as distributed memory. They contains also some Flips-Flops, but instead of LUTs it offers double ports memory and shift registers. Amid the others logic resources, there are inputs-outputs, clocks *(CMT : Clock Management Tile)* and every interconnections between the different logic blocks.

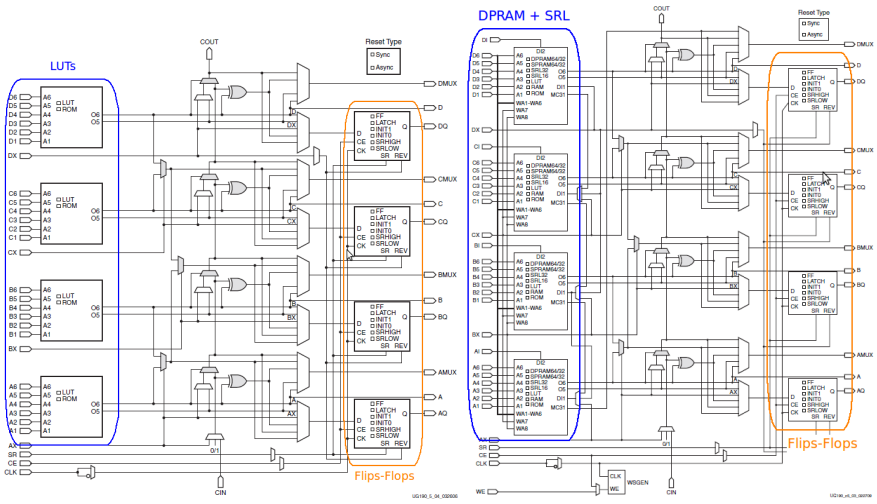


Figure 3.10: Slice-L and Slice-M [Xilinx 2009c]

[4]Look-Up Tables

As Slices can be very costly when used as memory components, FPGA matrices contain also physical Block RAM *(BRAM)*. On a maximum size of 36Kb, BRAMs are disposed in columns. A BRAM can either serve as storage memory or be used as FIFO.

Like memory, arithmetic operations are resource consuming when using logic blocks. This is why Digital Signal Processing *(DSP)* blocks have been added in Virtex 5 FPGA and allow to process arithmetic operation on 48-bits wide data.

3.3.1.1 FPGA organization

The FPGA is divided into rows and columns *(Fig. 3.11)*. Rows are numbered in ascending order from the center of the FPGA. There may be a maximum of 20 rows. Rows are indexed from 0 to 9 on each side. The height of each row is 20 CLBs, which corresponds to a column served by a global clock line *(HCLK)*.

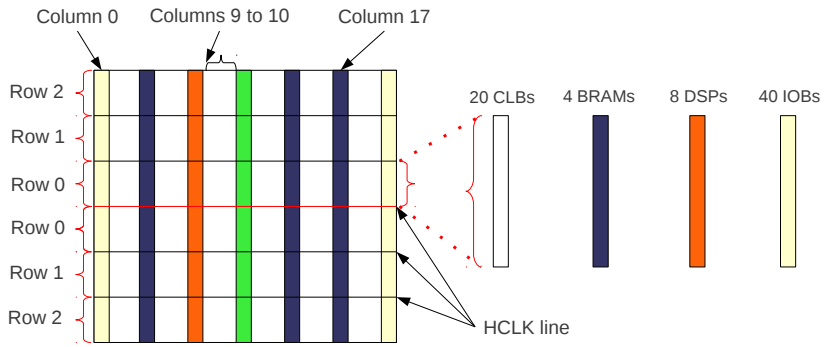


Figure 3.11: FPGA organization

3.3.2 FPGA configuration

A bitstream is a sequence of commands and data sent to the configuration port of the FPGA. If we look further in the understanding of the reconfiguration process, these commands are written in the registers of a configuration driver. In the case of the dynamic and partial reconfiguration, this driver is the ICAP driver.

3.3.2.1 Configuration port protocol

Configuration commands allow the user to read and write configuration data in the configuration memory, to control the reconfiguration process and to check if any error occurs during the configuration with the help of the CRC[5] register. Commands are sent as packets of 32-bits words. There are two types of packets:

[5]Cyclic Redundancy Check

- Packets of type 1 that are used to read and write configuration registers. It first sends a header in which we define the order to be achieved *(NOP, Read or Write)*, the register in which we wish to operate *(CRC, Control, Command, Address Frame, ...)* and the number of words we want to read or write. Then it sends the configuration memory data.

Header Type	Opcode	Register Address	Reserved	Word Count
[31:29]	[28:27]	[26:13]	[12:11]	[10:0]
001	xx	RRRRRRRRRxxxxx	RR	xxxxxxxxxxx

Notes:
1. "R" means the bit is not used and reserved for future use.

Figure 3.12: Type 1 Paquet Header Format [Xilinx 2009b]

- Type 2 packets that are used to send or receive larger blocks of data. A packet of type 1 has to be sent previously in order to specify the address where data has to be sent.

Header Type	Opcode	Word Count
[31:29]	[28:27]	[26:0]
010	RR	xxxxxxxxxxxxxxxxxxxxxxxxxxx

Figure 3.13: Type 2 Paquet Header Format [Xilinx 2009b]

3.3.2.2 Frame address

Among all the available commands, one of the most interesting regarding the preemption is the command which allows to specify the address where the configuration content is written. It means which area of the FPGA is targeted. An area is located using the address of its first frame. A frame is the smallest unit of reconfiguration of the FPGA. A frame is 1-bit width, and its height correspond to a row of the FPGA. A frame has a total of 1312 bits. Each frame has a unique 32 bits width address which is divided into five parts *(Fig. 3.14)*.

The first part *(bits 23-21)* represents the **Block Type**. In a Virtex FPGA, there are four types of blocks:

- the reconfigurable blocks and interconnect: it includes CLBs, IOBs, DSPs, BRAMs, and clocks
- the BRAMs content

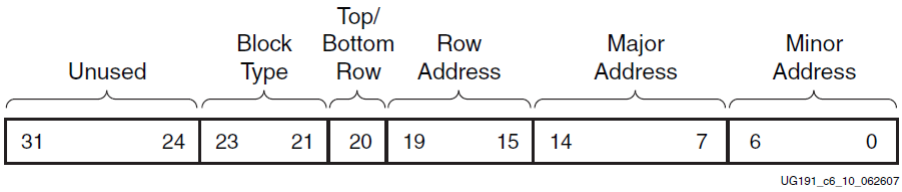


Figure 3.14: Frame address [Xilinx 2009b]

- the special blocks that are used for the partial and dynamic reconfiguration. In each column there is a special frame available at minor address 0. In this frame, the 21^{st} word corresponds to the HCLK, three of the four configuration bits following the 12-bits of the ECC *(error correcting code)* are used for example for the capture of registers.

- the BRAMs non-configuration blocks which yield device-specific data [Xilinx 2009b]

The second part is the **Top/Bottom Row** bit which indicates if the frame is located in the upper part of the FPGA *(Top = 0)* or in the lower one *(Top = 1)*. It should be noticed that except for the HCLK rows, it is necessary to reverse the bit order of the configuration frames to relocate a frame from the top to the bottom of the FPGA, and inversely.
Row Address: It corresponds to the row address as indicated in *Figure 3.11*.
Major Address: it corresponds to the column of resources that we want to reconfigure. In the case of the BRAMs, the configuration and the content is targeted by two different addresses.
Minor Address: it represents a frame inside a column of resources. These frames allow to access to the routing and logic configuration of each column, as shown in *Figure 3.15*.

The number of frames to write is dependent of the type of the targeted resources. However, each frame is composed of 41 words of 32 bits width. Figure 3.16 shows how a frame is composed along of a CLB row.

3.3.3 Bitstream parser

Beyond the need to know where are written the data, it is necessary to find this information inside a bitstream in order to be able to modify it on-line. This information will be useful to be able to relocate a module. To do so, we need to parse the partial bitstream and identify its content.

Each generated bitstream starts by a header indicating the design name, the FPGA part in which it has to be loaded, the time and date of creation, as well as the payload configuration data size *(Table 3.1)*.

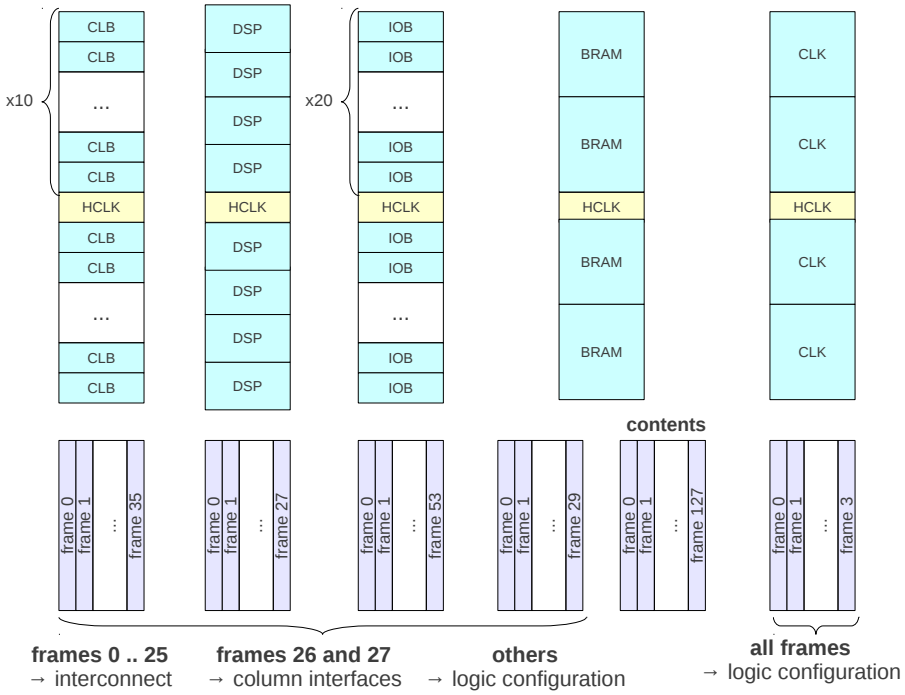


Figure 3.15: Resources memory configuration for the Virtex 5 architecture

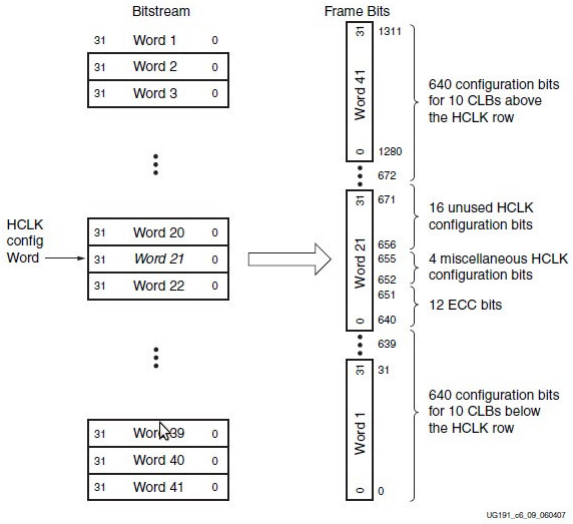


Figure 3.16: Frame composition [Xilinx 2009b]

3.3.3.1 Initialization commands

Each bitstream includes a first sequence of commands which permits to synchronize with the FPGA, to initialize the CRC register, and so on. This sequence differs regarding the bitstream type if it is a full or a partial bitstream. For a partial

Field name	Size in byte	Default value
Magic Number Length	2	0x0009
Magic Number	8	0x0FF00FF00FF00FF0
Null Character	1	0x00
Half-Word	2	0x0001
'a'	1	0x61
Design Name Length	2	—
Design Name	—	—
Character ';'	1	—
User ID	17	UserID=0xFFFFFFFF
Null Character	1	0x00
'b'	1	0x62
Part Name Length	2	—
Part Name	—	—
Null Character	1	0x00
'c'	1	0x63
Date Length	2	0x000B
Date	10	—
Null Character	1	0x00
'd'	1	0x64
Time Length	2	0x0009
Time	8	—
Null Character	1	0x00
'e'	1	0x65
Bitstream Length	4	—

Table 3.1: Bitstream header contents

bitstream, the sequence of commands is described in Table 3.2.

A part of this sequence which interests us is the one succeeding the writing of the *Write CFG* command. It starts by a writing in the FAR register *(Frame Address Register)* of the starting address of the reconfiguration. Followed by the number of words to write corresponding to the number of column to reconfigure in the addressed row.

3.3.3.2 Configuration data

The reconfiguration is done row by row. When switching from one row to another, a new address is sent in the FAR register and also the number of new words to write *(Fig. 3.17)*.

Command value	Description
0xFFFFFFFF	Dummy Word (x8)
0x000000BB	Bus Width Word
0x11220044	8 / 16 / 32 bus width
0xFFFFFFFF	Dummy Word
0xFFFFFFFF	Dummy Word
0xAA995566	Sync Word
0x20000000	No Operation (NOP)
0x30008001	Type 1 write 1 word to CMD register
0x00000007	Reset CRC
0x20000000	NOP
0x20000000	NOP
0x30018001	Type 1 write 1 word to ID register
0x02E9A093	ID Code
0x30008001	Type 1 write 1 word to CMD
0x00000001	Write CFG command
0x20000000	NOP
0x30002001	Type 1 write 1 word to FAR
0x00100780	FAR value
0x20000000	NOP
0x30004000	Type 1 write 0 word to FDRI
0x50002031	Type 2 write 8241 words to FDRI

Table 3.2: Bitstream initialization commands

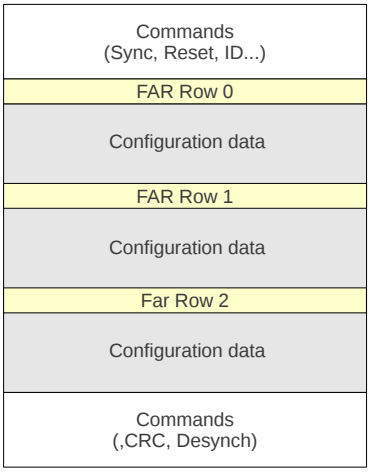


Figure 3.17: Multiple Rows bitstream content

For partial bitstreams, in addition to configuration data, an additional frame is

written. The presence of this frame is not documented but may correspond to an extra frame for synchronizing the HCLK row that is at the center of the frame. This frame is found at the end of each configuration data corresponding to a FPGA row.

3.3.3.3 Desynchronization commands

After the configuration data, a series of NOP *(No Operation)* is sent to allow the FPGA to complete the reconfiguration. This is followed by a control command of the CRC and an ICAP desynchronization.

Command value	Description
0x3000C001	Type 1 write 1 word to MASK register
0x00001000	CTL0 - Enable System Monitor Overtemperature Power Down
0x30030001	Type 1 write 1 word to CTL1 register
0x00000000	NULL
0x30008001	Type 1 write 1 word to CMD register
0x00000003	Last Frame (LFRM)
0x20000000	Dummy Word (x 101)
0x30002001	Type 1 write 1 word to FAR register
0x00ef8000	Row = 15, Top = 0, Block type = 7
0x30000001	Type 1 write 1 word to CRC register
0x6efece57	CRC value
0x30008001	Type 1 write 1 word to CMD register
0x0000000d	Desync
0x20000000	NOP

3.4 Preemption mechanisms

Preemption implies the ability for an operating system to save the execution context of a hardware thread running in a given slot, to load a new one into this slot, and later, to restore the context of the first hardware thread in any of the available slots of the platform. The last requirement involves to be able to relocate a hardware thread from one slot to another.

To manage hardware threads in this way, using the information from the previous section, we implement three operating system services which are a context management service, a reconfiguration service and a relocation service.

3.4.1 Context management service

There are two ways to save the context of a hardware thread. Either using checkpointing mechanisms [Huang 2008] or processing partial readback [Lee 2010]. The first one is intrusive and implies that the developer inserts checkpoints in his source code. Checkpoints are the only moments where the preemption is enabled. To

preempt a thread, the scheduler has to wait until the thread reaches a checkpoint and so saves its context. Consequences are a time overhead at each checkpoints and latency in preemption decision. The advantage is that the context size could be dramatically reduced, and ideally to zero.

The second way is the readback mechanism which is technology dependent but which avoids real time failure since preemption could be done immediately without risk to lose information. This is what we have chosen to use. Readback consists of reading the contents of the partial zone where the module is located. In the case of the hardware threads, segregation between static part and dynamic part permits task context reduction and offers a common interface in order to integrate different accelerators in the same partition. It should be noticed that application must ensure preemption is disabled when the thread is currently communicating to avoid blocking or data loss.

3.4.2 Reconfiguration service

A design using the partial and dynamic reconfiguration technique provided by Xilinx FPGAs [Lysaght 2006], is composed of a static part and several reconfigurable partitions in which reconfigurable modules can be loaded. Using this technology, the operating system is able to schedule hardware threads [Belaid 2009], without resetting the rest of the system. For real-time applications, both readback and reconfiguration overheads can be minimized using dedicated hardware reconfiguration controller, such as FaRM [Duhem 2011], Uparc [Bonamy 2012] or the solution offered by Hansen et al. [Hansen 2011]. For instance, FaRM, which is used in the design test detailed in Section 3.5.5, allows to process configuration with the theoretical maximum throughput of 400 MB/s at a frequency of 100 MHz.

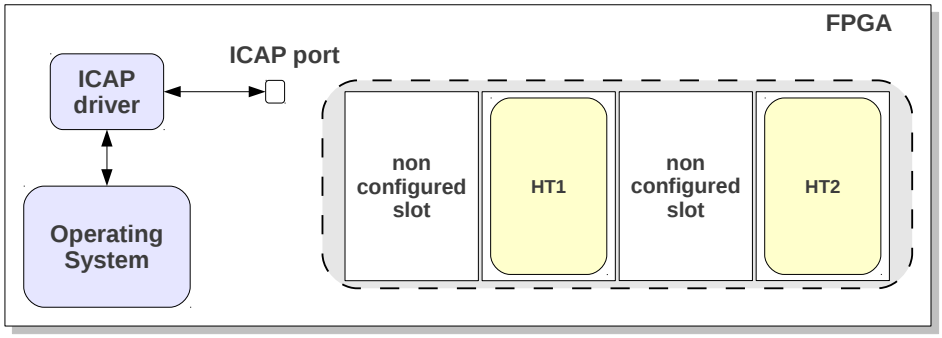


Figure 3.18: ICAP driver for Partial Reconfiguration

3.4.3 Relocation Service

As logic resources are critical in FPGAs, we would want to be able to run several threads in the same reconfigurable slot at different times. One of the issue encountered in the classical flow is that a partial bitstream for a given module is generated

for one slot and only one. To load a module on another slot, we need either another bitstream, which is memory consuming, or a relocated bitstream, whose creation is time consuming.

In embedded system, with the increase of the FPGAs size, and so of the bitstream size, the amount of memory needed to store one specific partial bitstream for each targeted partition is becoming more and more prohibitive. This is why a relocation service seems to be the best choice. To relocate a partial bitstream, we implemented two services: a bitstream parser and a bitstream relocater.

A bitstream parser is needed to find the right information in the bitstream. Xilinx FPGAs are organized in rows and columns. Each column is composed of several frames, which is the smallest reconfigurable entity. To reconfigure a FPGA, the ICAP reads a bitstream, writes address information in the Frame Address Register *(FAR)* of the ICAP and writes frames contents into FPGA memory. Information which interests us in the bitstream are the FAR values and the CRC value. The process to relocate a partial bitstream is detailed in *Fig. 3.19*.

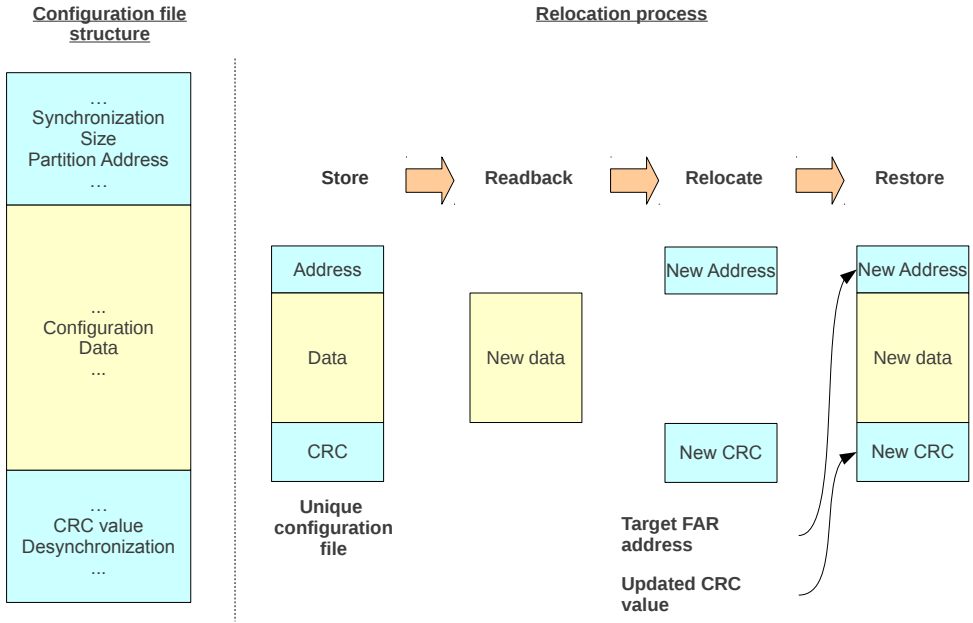


Figure 3.19: Partial bitstream relocation process

This process needs two bitstreams, one for the source partition and the other implemented for the target partition. A readback is done on the first partition. The resulting context is then saved in a new bitstream. The headers and footers of the second bitstream are then modified to target the wanted partition modifying the FAR value and adding a newly computed CRC value. In order to decrease the time overhead of the CRC computation, we implemented a dedicated hardware module. Details on this implementation can be found in Appendix B.2. Finally, merging

the headers and the saved context, we get a new relocated bitstream. A list of the different available partitions, identified by their FAR value and their size, can be created in order to simplify this process. This solution would make the second bitstream unnecessary.

3.5 Design flow for hardware threads relocation

3.5.1 Standard flow

In the latest versions of the Xilinx software ISE Design Suite *(IDS)*, the realization of a design composed of reconfigurable modules, but not relocatable, has been greatly simplified and automated for the end user.

3.5.1.1 Dynamic partitions

The user, from a complete static design, has the ability to create one or more dynamic partitions. In each of these partitions, it will be able to instantiate one or more modules. Modules implemented in the same partition have to share the same inputs and outputs, without necessarily using all of them. If a module need to be implemented on several partitions, one instance of this module is created for each partition *(Fig. 3.20)*.

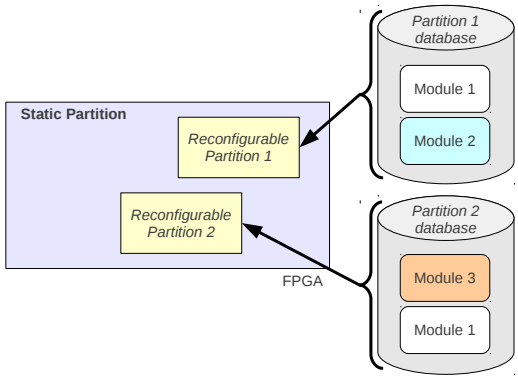


Figure 3.20: Partial reconfiguration: Partition and modules

3.5.1.2 Proxy Macros interconnections

Once each module dynamically assigned to a partition, the user has to place and route the static partition. At this moment, we end up with a design whose static part is placed and routed and which contains *"proxy macro"*, which are bus macros automatically placed by the tool *(Fig. 3.21)*. These *"proxy"* are placed in the dynamic partitions, at the boundaries with the static partition. They ensure that the inputs and outputs signals of the modules sharing the same partition go through

the same path and that there will be no routing issues during the reconfiguration of this partition.

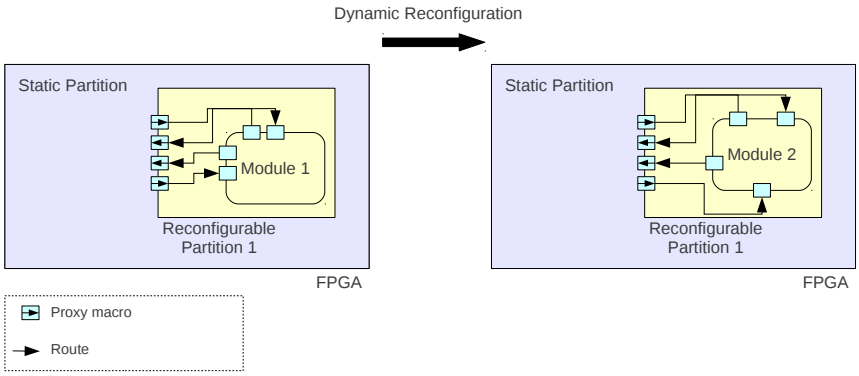


Figure 3.21: Proxy Macro Placed and Routed example

3.5.2 Problematics

Module relocation consists in moving a module from a dynamic partition to another. This process forces us to define relocatable modules. To do so, the following conditions are mandatory:

- the resources provided to a module should remain the same from one partition to another. Namely, partitions should be homogeneous regarding resources relative location. These resources include every logic block *(LUTs, BRAM, etc...)*, as well as the routing between these different blocks.
- the connection between the module and the static partition should be able to support the dynamic reconfiguration and should be homogeneous from one partition to another.

3.5.2.1 Partitions interconnection

The first issue which is encountered is that from a partition to another, even if they have identical shapes and have the same inputs and outputs, the *"proxy macro"* automatically generated for these inputs and outputs by Xilinx tools are not mandatory placed at the same relative location inside the partition.

Help with design constraints, it is possible to control where these proxy are placed but not the route between this proxy and the dynamic partition. Concretely, it is possible to place an input or output signal of the module, and so a proxy, on a given slice, but it is not possible to constrain which input or output of the slice will be used. From a partition to another, routing in each partition will be likely different and so the relocation of a module will lead to a routing failure.

To leverage this issue, several works listed in the state of the art used hard bus macro *(Slice macro in Figure 3.22)*. These hard macros are manually placed on both sides of the boundary between two modules, a static one and a dynamic one or both dynamics. Routing between the two modules is defined inside the macro and remains the same during the implementation.

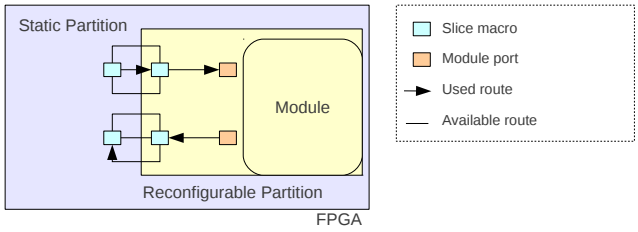


Figure 3.22: Slice Macro

It was possible, in the 10.1 version of PlanAhead and earlier, to manually add bus macro. In the latest version, the tool allows the user to define placement constraints which place the bus macro on the boundary between the static partition and the dynamic module. However, during the implementation, these constraints are not always respected and the macro could be moved in the static part as the tool keeps automatically placing its own proxy macro *(Fig. 3.23)*.

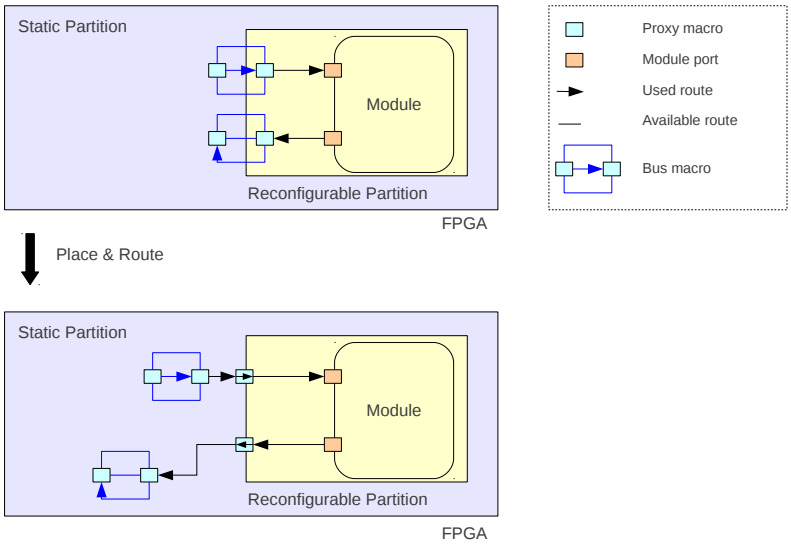


Figure 3.23: PlanAhead Slice Macro placement

It should be noticed that there is a way to suppress the automatic insertion of proxy macro setting the PARTITION_PIN_DIRECT_ROUTE constraint to *true*. This has the effect to route the partition output directly to the static partition

without inserting an additional proxy macro. However, the bus macro previously placed are always replaced somewhere inside the static partition.

This is due to the fact that the partial reconfiguration flow provided by PlanAhead do not support a dynamic part to be overlapped by a static one. A new flow must be found using only the static flow provided by PlanAhead and inserting additional constraints during the implementation of each module which would keep the required placement and prohibition constraints.

3.5.2.2 Partitions routing

The second issue which need to be solved regarding the relocation of dynamic modules is the partitions routing. The routing matrix of Xilinx FPGAs is known to be glitch-less. Namely, when the routing matrix is reconfigured, if there is no modifications to the configuration memory of a routing wire, this one will not be disturbed by the reconfiguration. As a consequence, routing wires of the static partition are allowed to go across the dynamic partitions without being affected by the dynamic reconfiguration process *(Fig. 3.24)*. This mechanism facilitates the routing process and improves its efficiency regarding timing constraints.

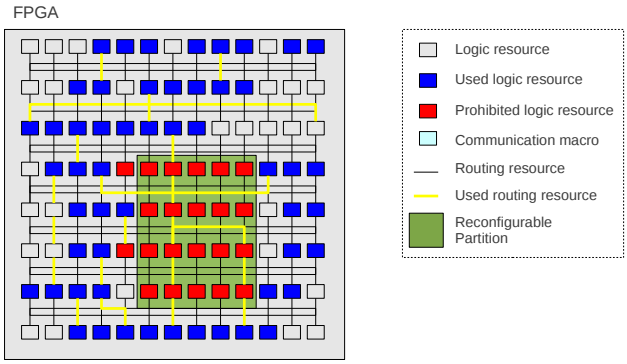


Figure 3.24: Static route through Reconfigurable Partition

The drawback when the objective is to be able to dynamically relocate the partial modules over every available partition is that when the module context will be saved, in addition to the module routing information, the static routing which go across this module will also be saved into this context. As a consequence, when relocating the module, the static routing of the target partition will likely not be the same than in the source one. So this routing may be cut and may produce a routing failure.

3.5.3 Relocation flow

We have to propose a new design flow which is able to apply the placement and prohibition constraints needed to design homogeneous and independent partitions. To deal with this issue, this section introduces the theoretical procedures steps of such a design flow and the possible solution offered by the alternative tools.

3.5.3.1 Procedure steps

The different steps to make a design supporting dynamic modules relocation are described in Figure 3.25:

1) *OFFLINE*: implementation of a static design containing empty slots to host the dynamic modules

2) *OFFLINE*: insertion of interconnection components between dynamic and static modules

3) *OFFLINE*: separate implementation of each dynamic module prohibiting the use of the static resources

4) *OFFLINE*: merging of the different implementations

5) *INLINE*: module execution stop and context save

6) *INLINE*: modification of the module context and computation of the new CRC value in order to place it in a new slot

7) *INLINE*: restoration of the module context and restart

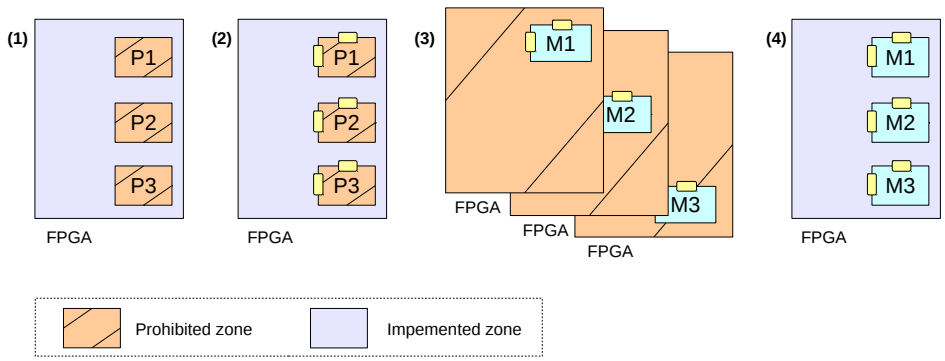


Figure 3.25: Relocation flow

3.5.3.2 Static partition design: Global methodology

As we cannot use the partial reconfiguration flow provided by PlanAhead as is, we have to use the static flow and add the necessary constraints to separate the dynamic modules from the static part of the system.

The first constraint to be applied is the placement constraint. Dynamic partitions can be delimited by prohibiting the placement of static resources. This prohibition is done using the CONFIG PROHIBIT constraint on the affected areas in the UCF constraint file. Unfortunately, this constraint prevents both the automatic placement of modules by the ISE placer, and the manual placement using the constraint file, so no bus macros can be added at this stage *Fig. 3.26*.

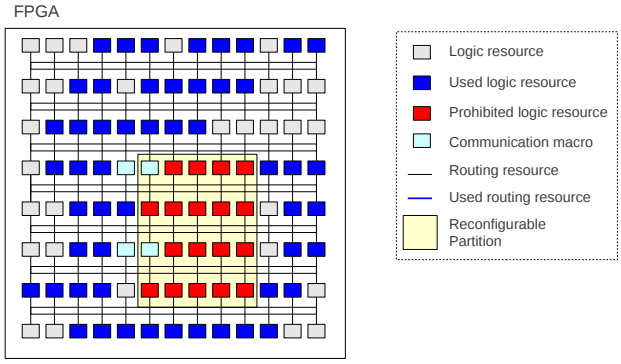


Figure 3.26: Static place

It is also necessary to apply routing constraints on dynamic areas to ensure that no wire from the static routing crosses the reconfigurable area. No constraints are defined in the Xilinx tools that would meet both needs. One solution is to use blocker macros, such as defined by Kock. et al. [Koch 2010a] [Koch 2009] in their design tool named *Recobus Builder*. In addition to block the routing, blocker macros occupy resources available in the dynamic area and so act as a placement constraint.

Macros can be generated using the XDL language *(Xilinx Description Language)*. This language allows to define the location of the different components *(internal and external signals of the modules)* on the circuit *(Slices, BRAM, DSP, ...)* and their configuration *(LUT used or multiplexer, values contained in the BRAM, ...)*. It can also describe the routing between these components, which will serve to block the routes within the reconfigurable areas.

The use of such a language permits to automate the generation of macros. The structure of the FPGAs matrix being regular at a certain level, it is possible to create communication bus composed of a same macro repeated along the boundaries, and to block areas of varying sizes.

3.5.3.3 Macro generation with XDL

An XDL file allows to describe a FPGA design at several levels of its implementation:

- after the placement step: the file describes the location of the different used resources, as well as how they are configured.
- after the routing step: in addition to placement information, the routing between the different resources is described using the semantics of the *Nets*.

The design is described in the file as a *Module*. This module has *Ports*, used to describe the input and output ports of the macro, as well as Slices, BRAMs, DSPs, and Nets.

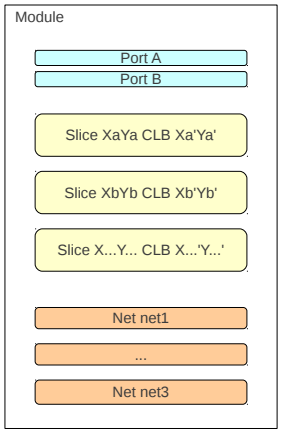


Figure 3.27: XDL File structure

The Slices are described specifying their coordinates, and the configurations of their LUTs, multiplexers and registers. These configurations indicate which inputs and outputs of the Slice are used, and what is the logic equation of the LUTs.

In addition to the resources dedicated to logic functions, global routing matrices are part of the FPGA resources. These matrices are located within each logic resource *(CLB, BRAM or DSP)*, and outside thereof.

Matrices provide access to internal inputs and outputs for each resource. Each input and output has a single routing path possible to the global routing matrix. External matrices offer more possibilities and allow to go to other global routing matrices in order to finally reach the internal matrix of the end point of the route *(Fig. 3.28)*.

The Nets are used to describe the interconnections between the various resources of a design. An interconnection is a route, which can propagate a signal from one point to another. In the XDL description, a route goes from the output of a

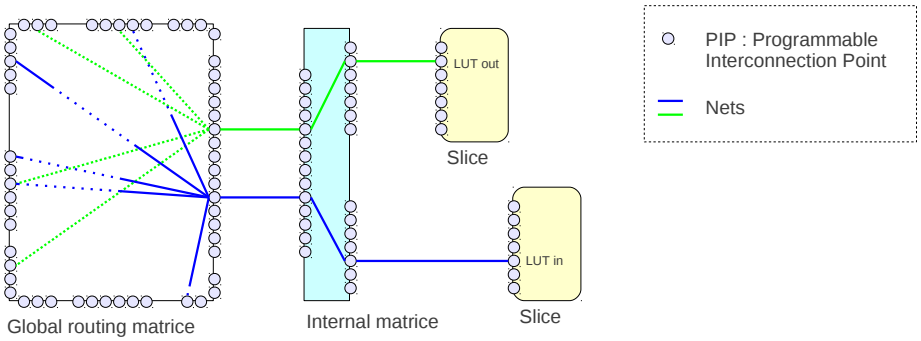


Figure 3.28: Internal and external switch matrices

resource to the input of several other resources. It passes through Programmable Interconnection Points *(PIPs)*. These PIPs are entry points and / or output of the routing matrices.

The transition from a global switch matrix to another follows a certain logic in the routing paths that can be taken from a given PIP. PIPs of external matrices are identified with a tag in their name. This tag permits to identify a pattern. There are three types of PIPs: PIPs starting a pattern *(BEG)*, intermediate PIPs *(MID)* and PIPs ending a pattern *(END)*. A pattern is a set of three matrices, whose the relative positions of the three types of PIPs, namely, BEG, MID and END, are identified by their name *(Fig. 3.29)*.

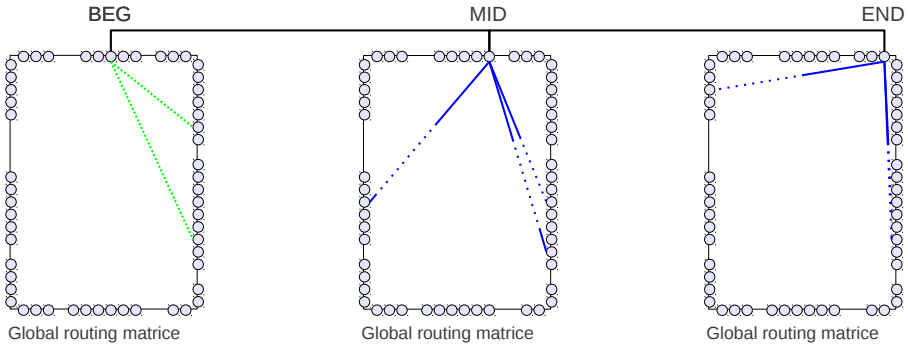
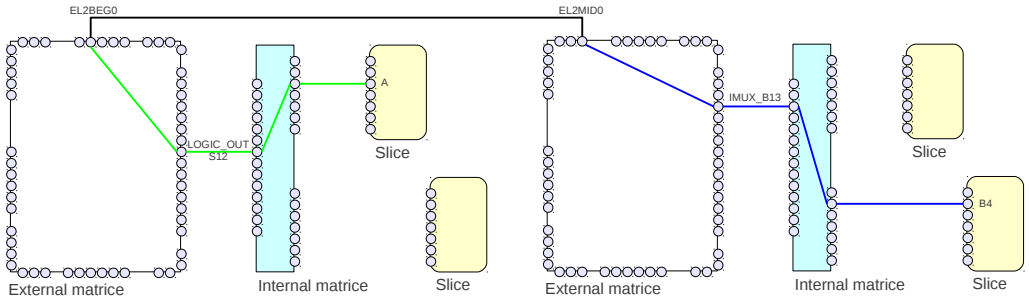


Figure 3.29: PIP types

In the XDL description, a PIP does not only refer to an interconnection point, but to a segment connecting two points of interconnection. The segments described relate only to internal segments, that is to say that the segments connecting two external matrices are implicitly described *(Fig. 3.30)*.

The XDL language is a good solution to control design routing at a very fine grain. Several of the alternative design tools presented in the next section turn on




```
net   "net_blocker_macro_CLBLM_X3Y0_S1_A"
      outpin  "blocker_macro_CLBLM_X3Y0_S1"  A  ,
      Pip  CLBLM_X3Y0  M_A  ->  SITE_LOGIC_OUTS12  ,
      Pip  INT_X3Y0  LOGIC_OUTS12  ->  EL2BEG0  ,

      inpin  "blocker_macro_CLBLL_X4Y0_S2"  B4  ,
      pip  INT_X4Y0  EL2MID0  ->  IMUX_B13  ,
      pip  CLBLL_X4Y0  SITE_IMUX_B13  ->  M_B4  ,
;
```

Figure 3.30: XDL Net example

the use of this language.

3.5.4 Experimented tools

3.5.4.1 RapidSmith

RapidSmith is a tool developed by the Brigham Youth University. It provides a framework to easily manage XDL files. It was designed in order to allow a developer to implement his own routing tools. In our case, it could be used to design custom macros, and especially it could help us to create blocker macros in order to apply fine routing constraints.

However, creating an efficient routing tools may take many time. Fortunately, another interesting tool called OpenPR has been realized. This tool is based on the same engine as RapidSmith and offers a higher level of abstraction in the design of independent partitions.

3.5.4.2 OpenPR

OpenPR is a tool based on the TORC [Steiner 2011] framework which provides a routing engine for the Xilinx FPGAs. OpenPR was created to offer the same features as those provided by the Xilinx Partial Reconfiguration Toolkit which was available for ISE 9.2, but not for the latest versions. It allows the design of empty dynamic region connected to the static partition through hardware bus macros. It is thought to be modular and above all extensible.

No IHM is provided with this tool and all manipulations are done in command line. A specific directory structure has been adopted to ease the choice of the implementable modules inside a given project. A project is defined by an XML file

in order to facilitate a future integration inside an IDE[6].

Using this tool, we managed to design dynamically reconfiguration applications on a Virtex 5 LX110 device. Unfortunately, depending on the complexity of the design, the routing constraints are not always respected, and the router can ignore them or keep stuck inside the place and route process. Nonetheless, this tool stays promising and its open-source characteristic makes it more flexible than a proprietary solution, even if more complex to implement.

3.5.4.3 Xilinx Isolation Design Flow

The Isolation Design Flow *(IDF)*, alternatively called Secure Chip Crypto *(SCC)* design flow, has been created to target fault-tolerant systems, especially in the critical applications in which safety and fault containment is a primary objective. This flow allows a designer to isolate the different modules of his system against each other. This is done regarding both the logic and the routing resources.

In this flow, each module to isolate is defined and synthesized separately. A top-level module groups all these modules as black boxes. To ensure a correct isolation, the implementation of these modules is done under some constraints. Namely, every connections between two isolated partitions have to pass through trusted routes *(Fig. 3.31)*.

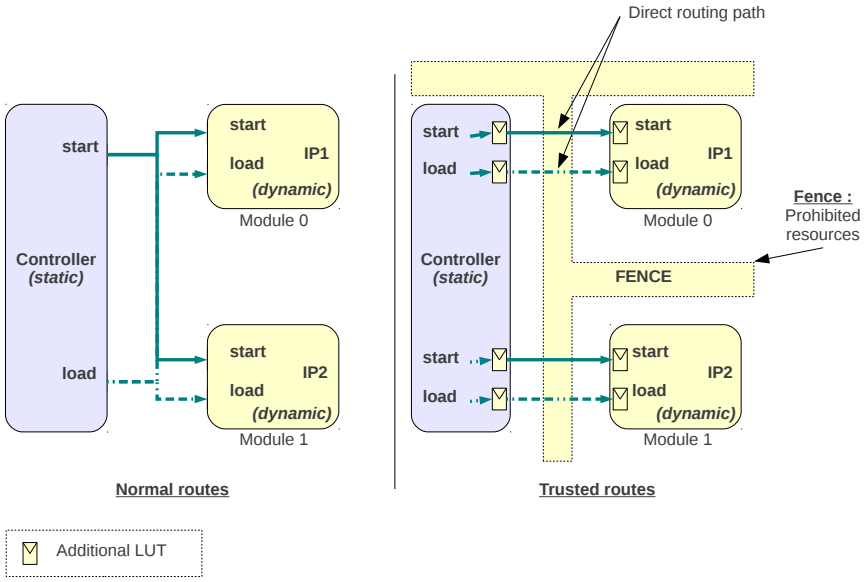


Figure 3.31: Trusted routes

A trusted route specifies that an output of a partition has to pass through a direct route. If the output is used as a load for two different inputs, this signal have to be split into two different signals passing through a LUT resource, and so

[6]Integrated Development Environment

forms what is called a trusted route. These constraints have to be applied to every inter-partitions signals when it is necessary except for the global signals such as the clock signal.

This flow has the advantage to be integrated into the PlanAhead tool provided by Xilinx and is available now for the Virtex-4, Virtex-5, and Spartan-6 devices and soon for the Kintex-7 devices. In the following, we choose to investigate the adaptation of this design flow in order to perform relocation on a Virtex-5 platform .

3.5.5 Adapted Isolation Design Flow

3.5.5.1 Hardware platform

Initially, the test of the Isolation Design Flow for the relocation of hardware module has been experimented on the simple design shown in *Fig. 3.32*, and implemented on a Virtex 5 SX50T FPGA using the version 13.1 of IDS[7]. It is a Microblaze-based platform composed of the FaRM IP described in Section 3.2.2 and used to reconfigure the dynamic partitions, a hardware CRC module used to compute the new CRC of the relocated module as well as two dynamic modules.

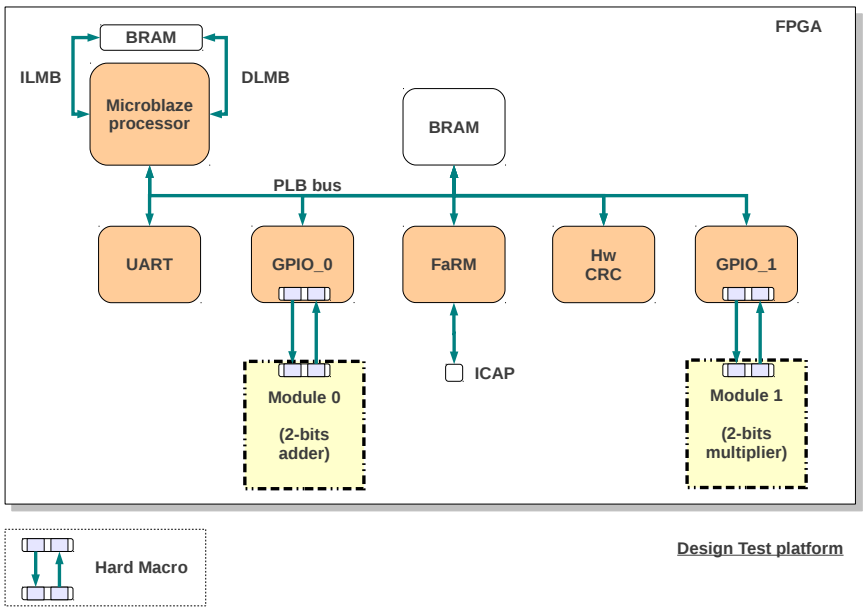


Figure 3.32: Test design

There is no external memory. The only off-chip connections are the FPGA clock and the reset button. The two reconfigurable modules implement respectively a

[7]ISE Design Suite

two-bits adder and a two-bits multiplier. Each one of these modules is controlled by the processor through a dedicated GPIO peripheral.

In the case of the relocation where routes between the static partition and the dynamic ones have to be relatively identical, we instantiated hard macros to connect these two types of partition.

3.5.5.2 Modules input and output signals

The Isolation Design Flow requires us to synthesize each module of the design separately. In each isolated module, inputs and outputs which are not directly connected to an input or an output pad of the FPGA has to be instantiated as a trusted route and therefore has to be defined in the HDL file as a non-buffered port as follow:

attribute buffer_type: string;
attribute buffer_type of <port_name> : signal is "none";

In order to improve the clocking routing of the design, the instantiation of the clock buffer has been removed from the top level source file and let to the control of the synthesizer. In this way, reconfigurable modules, like the static partition, use the global clock tree instead of a trusted route using a combinatorial path via a look-up table.

3.5.5.3 Software bus macro

Once each module is synthesized, the main part of the flow is done using the PlanAhead tool. Modules netlists are imported in the design and these which need to be isolated are converted into partitions. Each partition is configured with the SCC_ISOLATION attribute, which notifies that the partitions have to be designed using the Isolation Design Flow.

Then the physical block of each module is placed inside the FPGA matrix. Another constraint imposed by the Isolation Design Flow is that the inputs and the outputs pad used by a partition have to be included inside the region covered by its corresponding physical block. In addition, the boundary between two isolated partition have to be of at least one CLB-wide, horizontally or vertically. This boundary is called a *Fence* and is an area in which neither the logic resources nor the routing switch matrices will be used *(Fig. 3.31)*.

Firstly, as we wanted to provide a flexible solution for the instantiation of the bus macro, we let the routing engine of ISE creating the trusted routes. Therefore we relied on a software implementation of the bus macro.

To do so, we instantiated LUTs in the top-level source code to connect the reconfigurable partitions with the static one. For each wire of the bus macro, the

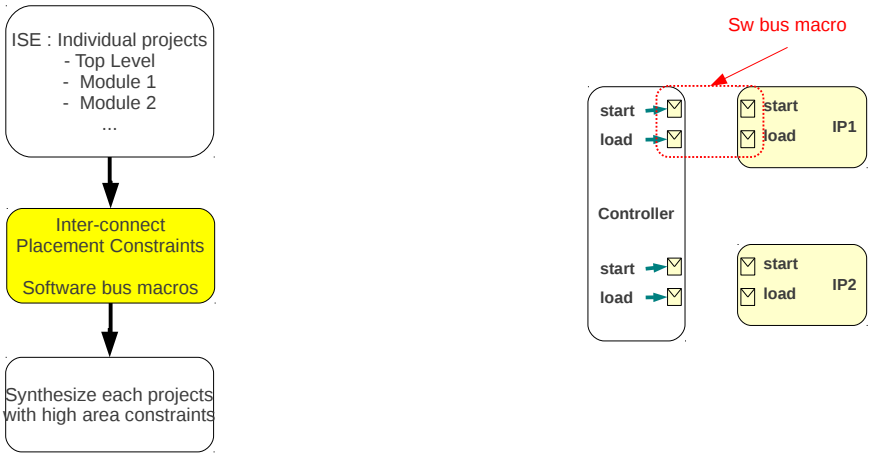


Figure 3.33: Software Bus Macro implementation

following constraints has been applied:

```
1  ------------------------------------------------------------------------
2  -- xps_gpio_0
3  -- bus macro LUTs outputs from static to dynamic
4     attribute LOCK_PINS of lut_xps_gpio_0_GPIO_IO_O_bm_in_0 : label is "
          ALL";
5     attribute LOCK_PINS of lut_xps_gpio_0_GPIO_IO_O_bm_out_0 : label is
          "ALL";
6
7     signal xps_gpio_0_GPIO_IO_O_bm_in_0 : std_logic := '0';
8     attribute S of xps_gpio_0_GPIO_IO_O_bm_in_0 : signal is "TRUE";
9
10    signal xps_gpio_0_GPIO_IO_O_bm_s_0 : std_logic := '0';
11    attribute S of xps_gpio_0_GPIO_IO_O_bm_s_0 : signal is "TRUE";
12
13    signal xps_gpio_0_GPIO_IO_O_bm_out_0 : std_logic := '0';
14    ttribute S of xps_gpio_0_GPIO_IO_O_bm_out_0 : signal is "TRUE";
15 --
```

And the instances of each LUTs:

```
1  -- input bus macro LUTs
2  lut_xps_gpio_1_GPIO_IO_I_bm_in_0 : LUT1
3  generic map (INIT => X"2")
4  port map (I0 => xps_gpio_1_GPIO_IO_I_bm_in_0, O =>
       xps_gpio_1_GPIO_IO_I_bm_s_0);
5  --
6
7  -- output bus macro LUTs
8  lut_xps_gpio_1_GPIO_IO_I_bm_out_0 : LUT1
9  generic map (INIT => X"2")
10 port map (I0 => xps_gpio_1_GPIO_IO_I_bm_s_0, O =>
       xps_gpio_1_GPIO_IO_I_bm_out_0);
11 --
```

The UCF file fixes the additional location constraints:

```
1  # input bus macro LUTs
```

```
2  INST "lut_xps_gpio_1_GPIO_IO_I_bm_in_0" LOC = SLICE_X**Y** | BEL = *6
       LUT;
3  #—
```

As a result, the constraints were too lazy and are not necessary respected by the synthesizer, so the routing between the two partitions can be implemented in several ways, even if the LUTs placement is respected *(Fig. 3.34)*.

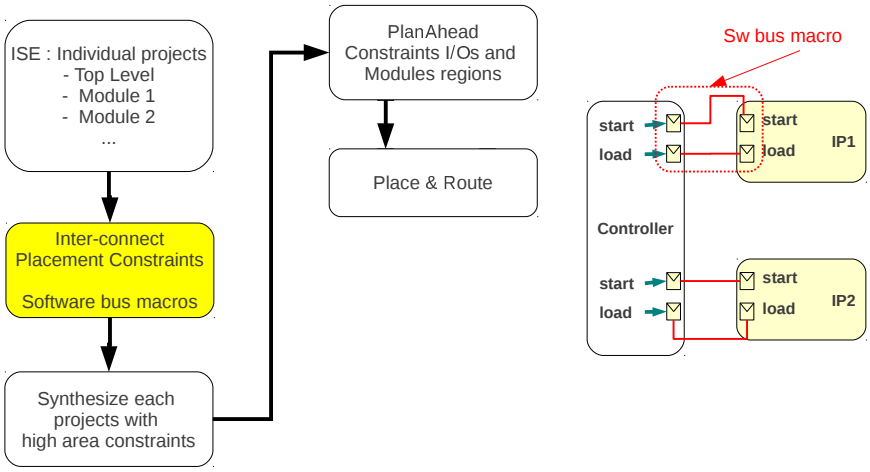


Figure 3.34: Routed software Bus Macro

3.5.5.4 Hardware bus macro

To overcome this issue we decided to implement hardware bus macros *(Fig. 3.35)*.

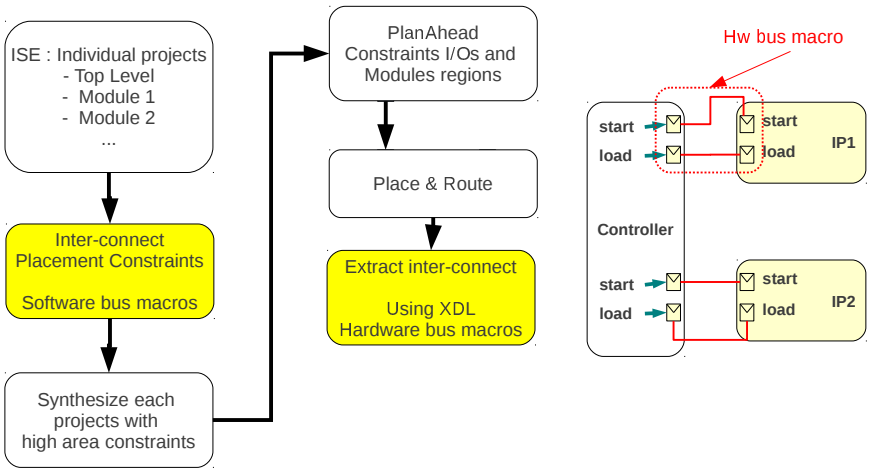


Figure 3.35: Hardware Bus Macro extraction

To create a hardware bus macro, we started to get the XDL description of the current implemented design. In this description, we looked for the LUTs which form the soft bus macro of the first partition. A first implementation with a soft bus macro is necessary before extracting a hard macro. These LUTs are then copied into a new XDL file and formatted to create a hard macro. The XDL description of the implemented design is obtained using the following command:

$ xdl ncd2xdl config_1_routed.ncd

where "config_1_routed.ncd" is the Native Description Circuit File generated after the place and route phase.

In order to extract and generate the hardware macro, we rely on the RapidSmith framework. More information, especially the installation process can be found on this website: http://rapidsmith.sourceforge.net/.

In our case, we limit ourself in the definition of hardware macros. To do so, we create a new project in the Eclipse framework. The application Lut_Macro_Extractor which is in the RapidSmith workspace allows to extract a software bus macro from a complete design and to create a new module which can be implemented as a hardware macro. The following listing gives a partial example of the generated macro for a bus macro named *bus_macro_v5_4io_tb* which contains seven inputs and seven outputs, defined with generic names.

```
1  design "__XILINX_NMC_MACRO" xc5vsx50tff1136-1;
2
3  module "bus_macro_v5_4io_tb" "bus_macro_v5_4io_tb_0" , cfg "
       _SYSTEM_MACRO::FALSE" ;
4
5    port input0_tb "bus_macro_v5_4io_tb_0" "D1" ;
6    port input1_tb "bus_macro_v5_4io_tb_0" "C1" ;
7    port input2_tb "bus_macro_v5_4io_tb_1" "D1" ;
8    port input3_tb "bus_macro_v5_4io_tb_1" "C1" ;
9    port input4_tb "bus_macro_v5_4io_tb_4" "D1" ;
10   port input5_tb "bus_macro_v5_4io_tb_4" "C1" ;
11   port input6_tb "bus_macro_v5_4io_tb_5" "D1" ;
12
13   port output0_tb "bus_macro_v5_4io_tb_2" "D" ;
14   port output1_tb "bus_macro_v5_4io_tb_2" "C" ;
15   port output2_tb "bus_macro_v5_4io_tb_3" "D" ;
16   port output3_tb "bus_macro_v5_4io_tb_3" "C" ;
17   port output4_tb "bus_macro_v5_4io_tb_6" "D" ;
18   port output5_tb "bus_macro_v5_4io_tb_6" "C" ;
19   port output6_tb "bus_macro_v5_4io_tb_7" "D" ;
```

The hardware macro is composed of several LUTs. A LUT component is defined and its inputs and outputs are configured regarding the signals which are routed through this LUT:

```
inst "xps_gpio_0_GPIO_IO_I_bm_s_0" "SLICEL",placed CLBLM_X34Y19
    SLICE_X47Y19  ,
  cfg " A5LUT::#OFF A6LUT::#OFF ACY0::#OFF AFF::#OFF
  AFFINIT::#OFF AFFMUX::#OFF
  ...
  D5LUT::#OFF D6LUT:lut_xps_gpio_0_GPIO_IO_I_bm_in_0:#LUT:O6=A1
  _BEL_PROP::D6LUT:BEL:D6LUT DCY0::#OFF DFF::#OFF
  DFFINIT::#OFF DFFMUX::#OFF DFFSR::#OFF DOUTMUX::#OFF
  DUSED::0 PRECYINIT::#OFF REVUSED::#OFF SRUSED::#OFF
  SYNC_ATTR::#OFF "
;
```

The next listing illustrates one of the extracted interconnect:

```
  net "xps_gpio_0_GPIO_IO_I_bm_s_0" ,
  outpin "xps_gpio_0_GPIO_IO_I_bm_s_0" D ,
  inpin "xps_gpio_0_GPIO_IO_I_bm_out_0" D1 ,
  pip CLBLM_X34Y19 L_D -> SITE_LOGIC_OUTS11 ,
  pip CLBLM_X34Y21 SITE_IMUX_B42 -> L_D1 ,
  pip INT_X34Y19 LOGIC_OUTS11 -> NL2BEG_S0 ,
  pip INT_X34Y21 CTRL2 -> CTRL_BOUNCE2 ,
  pip INT_X34Y21 CTRL_BOUNCE2 -> IMUX_B42 ,
  pip INT_X34Y21 FAN3 -> FAN_BOUNCE3 ,
  pip INT_X34Y21 FAN_BOUNCE3 -> CTRL2 ,
  pip INT_X34Y21 NL2MID0 -> WL2BEG1 ,
  pip INT_X34Y21 WL2BEG1 -> FAN3 ,
  ;
```

This interconnect is translated to fit with the generic names of the inputs and outputs. It becomes:

```
net "xps_gpio_0_GPIO_IO_I_bm_s_0" ,
  outpin "bus_macro_v5_4io_bt_0" D ,
  inpin "bus_macro_v5_4io_bt_2" D1 ,
  pip CLBLM_X34Y19 L_D -> SITE_LOGIC_OUTS11 ,
  pip CLBLM_X34Y21 SITE_IMUX_B42 -> L_D1 ,
  pip INT_X34Y19 LOGIC_OUTS11 -> NL2BEG_S0 ,
  pip INT_X34Y21 CTRL2 -> CTRL_BOUNCE2 ,
  pip INT_X34Y21 CTRL_BOUNCE2 -> IMUX_B42 ,
  pip INT_X34Y21 FAN3 -> FAN_BOUNCE3 ,
  pip INT_X34Y21 FAN_BOUNCE3 -> CTRL2 ,
  pip INT_X34Y21 NL2MID0 -> WL2BEG1 ,
  pip INT_X34Y21 WL2BEG1 -> FAN3 ,
  ;
```

At the end, we get a design whose static-dynamic interconnection are relatively homogeneous:

3.5.5.5 Design synthesis

An issue which occurs when implementing these hard macro is that the placement constraints is not respected in the sense that the macro is systematically moved during the placement phase. It is an issue due to the fact that this macro overlaps both the static and the dynamic areas.

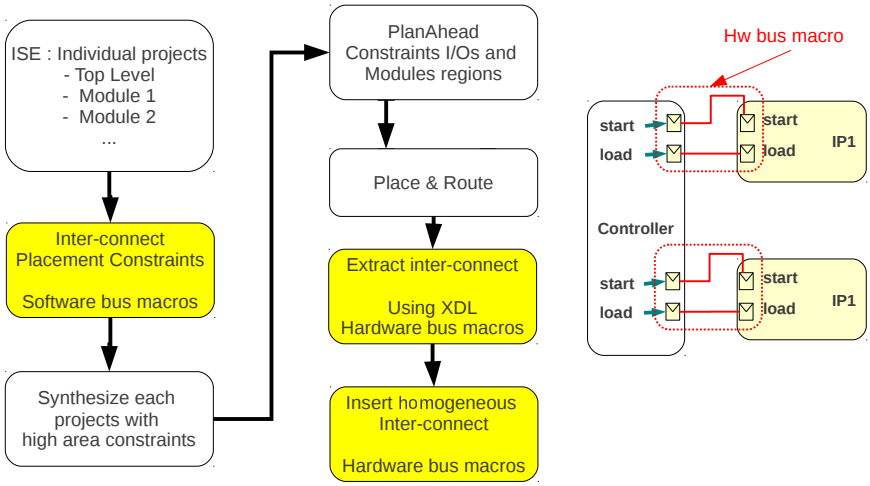


Figure 3.36: Hardware Bus Macro extraction and homogenization

To overcome this issue, the following location constraints applied to the hard macros have to be inserted in an external constraint file and passed to the XST synthesizer using the *-uc* flag:

$$INST <hard\_macro\_name> LOC = SLICE\_X\#Y\#;$$

where "#" represents valid Slice X and Y coordinates. This flag ensures that the synthesizer will respect the location constraints and that the hard macro will be placed at the correct position, over the static and the dynamic boundary. Finally, once the design is placed and routed, the correct isolation of each partition can be checked with the help of the Isolation Verification Tool *(IVT)* [Corbett 2012] and partial and full bitsreams can be generated *(Fig. 3.37)*.

After implementation *(Fig. 3.38)*, the two modules are well isolated in terms of logic and routing resources, and the one CLB-wide boundary between the dynamic modules and the static partition is respected. This successful result permitted us to perform a safe relocation of these two modules in the available dynamic partitions without additional bitstreams and, the most important, for the first time with the standard current design tools provided by Xilinx.

3.6 Conclusion

In this chapter, we introduced tools and mechanisms which allow us to manage the hardware threads like their software counterparts. Starting from here, we are able to provide at the operating system level, an API to create, delete or preempt hardware threads. All these features can serve as a basis for a hardware threads management service which can be integrated into an operating system.

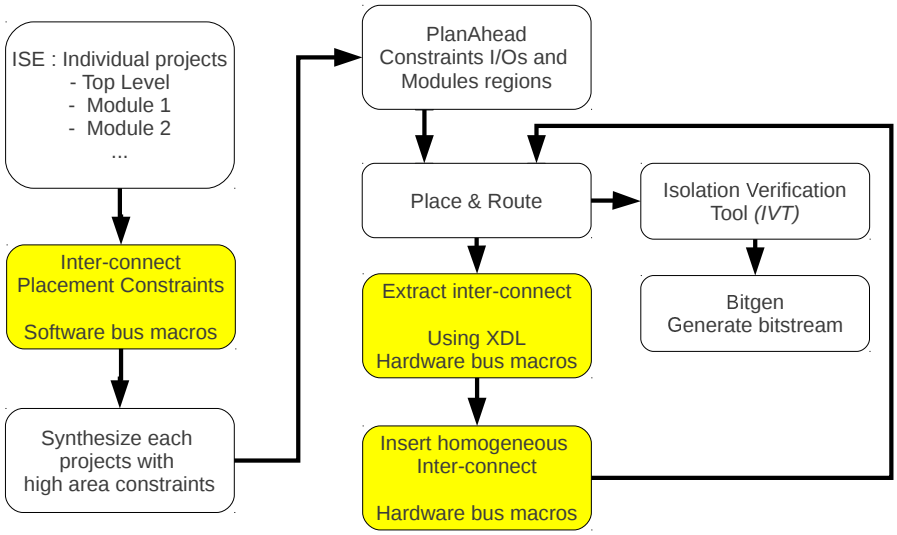


Figure 3.37: Adapted Isolation Design Flow

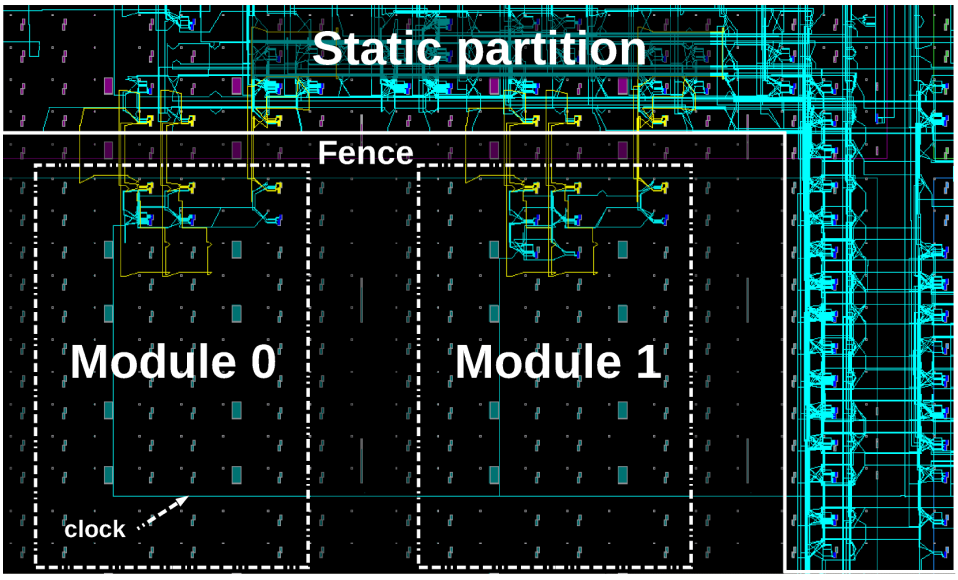


Figure 3.38: Design test - Partition isolation

In spite of its useful multi-processor communication layer, the RTEMS operating system chosen in the frame of the FOSFOR project is not flexible enough to match more precisely with the flexibility provided by the reconfigurable platforms. The MPCI prevent the user to create distant resources and a thread which would want to remotely access to a given resource, have to run on a core which implement itself the service able to manage the resource. For instance, if a hardware thread wants to access to a memory partition, the partition service should be implemented in the

hardware operating system. Regarding the specificity of each core, some services are more suitable to be implemented on one core rather than on another one.

For these reasons, in the next chapter, we go a step further than in the FOSFOR project and deal with the specification of a new operating system dedicated to the heterogeneous reconfigurable platforms. This operating system would be able to abstract the heterogeneity and so to offer the same API to handle the hardware threads than the one used to managed the software ones. In order to handle the heterogeneity in a flexible way, this management has to be extended and include the access to all available services of the platform.

Chapter 4

Operating System for Dynamically and Reconfigurable Heterogeneous SoC

Contents

4.1 Context and definitions

Heterogeneous Reconfigurable Systems-on-Chip allow us to implement an application with software threads and hardware threads. In this chapter, our objective is to facilitate the cohabitation between these heterogeneous entities.

As seen in the introduction of Chapter 2, software threads are managed on top of an operating system and therefore can access to the services it offers. Due to a need of scalability, an operating system dedicated to HRSoC should be distributed over the different cores composing this platform. However, in most of distributed operating systems, the services implemented on a core by an operating system can only be accessed by the threads running on this same core. In order to ensure a fair access to all services to every threads, both software or hardware, we have to provide a flexible solution which would extend the number of available services for all threads to a set of services which can be located on different cores. Such a solution should allow a flexible implementation of the operating system services over the platform, regarding the affinity each one has to run on a given core.

To be able to enforce this flexibility in an operating system, it is essential to correctly define the kernel structure of this operating system, as well as how the threads managed by this operating system will communicate, namely which inter-process communication API is implemented and how it is implemented for this purpose.

4.1.1 Kernel structure

The kernel structure is an essential element of an operating system. It defines the stability, the modularity, and the portability of a system. Currently, there are three types of kernel in the state of the art: monolithic kernel, micro-kernel and exo-kernel.

4.1.1.1 Monolithic kernel

Monolithic kernels, because of their conception in one and unique block, are the most performing kernels but also the less flexible. The portability and the maintenance of such a kernel is rather difficult and its structure is rarely well suited to handle the scalability and the adaptability required by multicore embedded systems.

4.1.1.2 Micro-kernel

Micro-kernels principle is based on the client-server model. Communication service between these two entities is realized through Inter-Process Communication *(IPC)* mechanisms. Regarding the performances, the first micro-kernels were very low compared to monolithic kernels.

Like for the thread model, the trend to fill this gap was the design of hybrid micro-kernel. The critical services like the memory management, the scheduler and the inter-process communication were bring back into the kernel.

This problem of performances has been imputed to IPC. Initiatives like the Mach [Accetta 1986] or L4 kernels [Liedtke 2001] allowed to reduce the impact of the IPCs in term of overheads. Actually, this issue is posed essentially when porting the kernel on processors like the x86, which have a protected mode needing a virtual memory manager. The use of micro-kernels as it has been though at the beginning, namely fully modular is possible in embedded architectures with a unified memory space. In all cases, the IPCs must be implemented carefully in order to achieve good performances.

4.1.1.3 Exo-kernel

The term of exo-kernel was invented by the Laboratory for Computer Science *(LCS)* at MIT[1]. Exo-kernels are actually micro-kernels pushed to limit. The role of the kernel is limited to the arbitration of accesses to material resources. The resources abstraction is minimal at the kernel level and customized at the user one. Services are known as libraries and are dynamically linked to the application level to let the user choose its own level of abstraction.

The advantage of this kind of kernel lies in the fact that almost all the operations are performed at the user-level, so the number of switches into kernel model is reduced to a minimum.

4.1.2 Thread API

This section draw a non-exhaustive list of existing standards for the C language and some specifications of software application programming interfaces *(API)* dedicated to operating system communication mechanisms.

4.1.2.1 MISRA-C

It is a standard used in the automotive *(MISRA = Motor Industry Software Reliability Association)*. This is a list of rules and guidance of "good" programming. The objective is to allow the developer to write portable and safe code.

4.1.2.2 POSIX

POSIX stands for Portable Operating System Interface and is a C standard. This standard will be portable and is widely used in the UNIX world. One of the interesting elements that may well apply to the embedded domain is thread management and all real-time extensions made in version 1003.1b-1993.

4.1.2.3 ARINC 653

The ARINC 653 is a specification of interfaces between the operating system and the application. This specification has been defined for the avionics. The operating

[1] Massachusetts Institute of Technology

system kernel is composed of two parts. The first part is a main module allowing to protect and multiplex the hardware resources. The second is specific to each system partition.

A partition is a subset of the operating system whose physical limitations were clearly defined. Each partition is independent, which provides a safe operation since the different address spaces are separated. A failure in one partition does not affect other partitions. The following services are defined:

- services related to partition management: creation, deletion, suspension, and completion of a partition
- the services of inter-partition communication: Message Queues and Sampled Message Queues
- services related to the multi-threading management, which corresponds to the intra-partition communication: Messages Queues, Sampled Message Queues, Events and Semaphores
- thread management
- failure management
- time management

Regarding interfaces and memory management, the developer does what he wishes when implementing this specification.

4.1.2.4 TIPC : Transparent IPC

Originally developed by Ericsson, TIPC is a communication protocol developed by VxWorks and now set free on Sourceforge. This is a protocol especially dedicated to networked systems. Each node of the system has a network address *(denoted N)*. These nodes are grouped into clusters *(denoted C)*. Finally, these clusters are themselves grouped into zones *(noted Z)*. A node address consists of the following: "Z_ID"."C_ID"."N_ID". Other features include:

- message size from the application point of view is between 1 and 66000 bytes
- synchronization between two ports is done by handshake
- there is a naming service to translate the name of a node address
- it also implements an error handler that manages transmission errors, inaccessible links and invalid names and addresses

Another feature, the same node can have multiple addresses which allows application to easily implement multicast by providing the same address to all subscribers of a channel.

4.1.2.5 MCAPI : Multicore API

The Multicore Association [Association 2012] is an association grouping industrial and academic partners with the aim of defining a new standard allowing to abstract communications among heterogeneous multicore platforms.

MCAPI for Multicore Communication API, is one of the three working groups of the Multicore Association. Its objective is to define a message-passing API to manage communications and synchronization between cores. The second working group, named MRAPI, is responsible for defining an API to manage resources which are shared by the processing elements of a heterogeneous multicore platform. MTAPI is a third working group which aims to define a new standard for thread management *(creation, placement, scheduling, ...)*.

These three working groups are the most important but there are other groups such as the Multicore Programming Practices Working Group and the Multicore Virtualization Working Group. Currently, a complete version of the MCAPI specification and a first stable one of the MRAPI specification have been released.

4.1.2.6 LINX

LINX for Linux is an open-source implementation of the LINX inter-process communication protocol. It targets heterogeneous multicore systems using the Ethernet protocol to communicate. On each of node, a thread is created and serves as a connection point *(proxy)* with the other nodes of the system. The API enables LINX to abstract the location of other threads running in the system and thus makes inter-process communication transparent to the user regarding the hardware platform used.

In addition it implements a neighbor discovery mechanism and handles cases where a server *(ie. another thread)* previously found, is no longer accessible. However, as it relies on the Ethernet protocol, this solution is too heavy for intra-chip communication.

Related works listed in the next section refers to some examples of what could be done to enhance the communication between threads in multicore and heterogeneous platforms. The choice of an well-adapted kernel structure and of a standard inter-thread communication API would permit to reach a good trade-off between performances and portability.

4.2 Related works

4.2.1 Introduction

In this section, we address the communication issue between threads located on different processors. Our final goal remains to leverage the heterogeneity of the platform, especially the fact that some services could be implemented more or less easily on a core. In this context,we are interested in providing the support of future design exploration tools which permit to find which service should be implemented

on which core. This support consists in offering to an operating system the ability to share its services with other threads running on remote processing elements.

Targeting multiprocessor system is an old issue. Several papers addressed this issue and especially the concerns of the multiprocessor communication. In these cases, a processor is considered a core. Protocols based on the shared memory paradigm, the message passing protocol or solution adapted from the cloud computing is presented in the following.

4.2.2 Inter-core communication in MPSoC

In [Tomiyama 2008] the authors propose an operating system for asymmetric multicore systems called TOPPERS-FMP *(Fig. 4.1)*. This operating system is based on the μITRON specification and its main characteristics are the system calls virtualization (for both local and remote calls), an independent execution for each node and a known limit for the inter-task communications latency in the worst case.

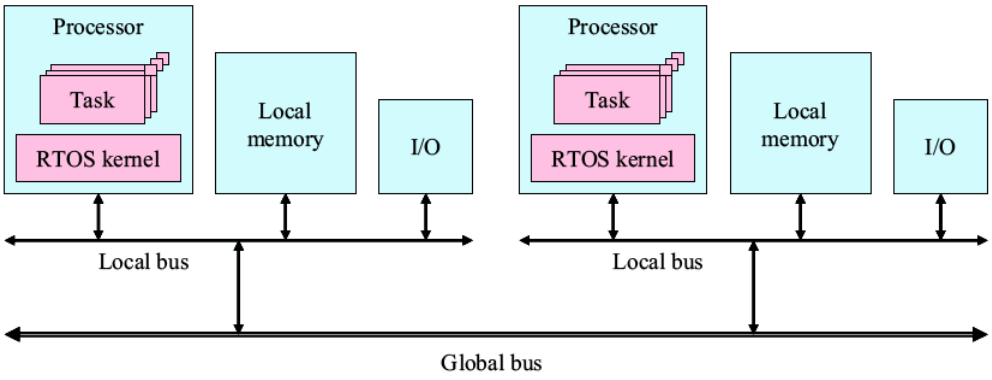


Figure 4.1: Toppers/FMP [Tomiyama 2008]

Following the μITRON specification, objects are classified: tasks and handlers are assigned to a processor and each one has a local scheduler, which guarantees independence between cores. Each object is identified by a unique identifier.

In this operating system, the inter-core communication mechanism is called IPSC for Inter-Processor System Call. Through this mechanism, a process has a direct access to the memory of another one due to the presence of a unified memory space in the shared memory. The synchronization between the two processors is done using spin locks[2] whereas intra-processor synchronization is realized by disabling interrupts.

The authors of [Huerta 2008] introduced a Symmetric MutliProcessor *(SMP)* based system composed of Microblaze processors *(Fig. 4.2)*. The operating system

[2] busy waiting for a lock

is spread over the platform and is in charge of task allocation on every processor. Task scheduling is globally manage by a unique scheduler. Communication between processors is realized via hardware interrupt mechanisms. This solution is easy to implement but lacks of scalability.

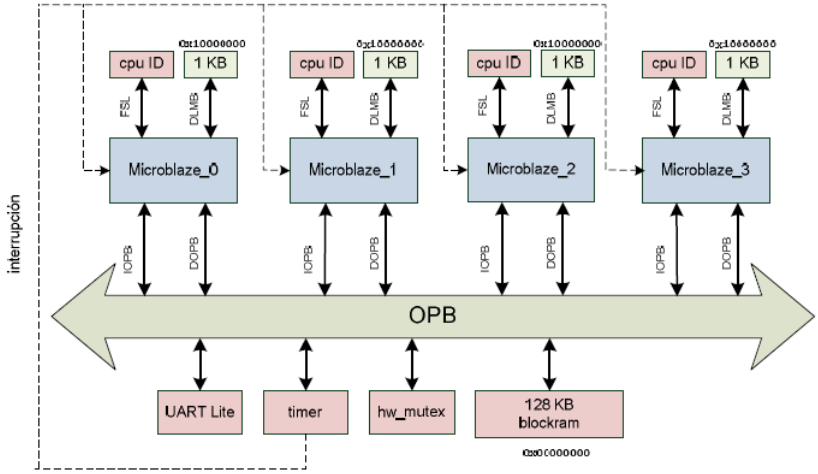


Figure 4.2: SMP System [Huerta 2008]

The subject of [Lin 2009] deals with Inter-Processor Communication *(IPC)* in heterogeneous multicore platforms. They aim to reduce the overhead due to the communication in periodic pipelined multicore applications.

They argue that classical protocols like Message Box, FIFOs or shared memory are badly linked to the monocore historical context. So they introduce the NTU[3] Inter-Core Process Communication *(NTU ICPC)*. This is a user-level protocol based on the sender-receiver paradigm. It is implemented at the middleware level. Its main goal is to limit the number of copy to one per transaction. It can be seen as a synchronous shared-buffer communication. This method has the advantage to avoid context switch and to improve the portability.

By default, communications use local buffers (*(called individual working space)*, otherwise if the sender and the receiver can both access to a shared buffer, they do a źero-copy IPC. This is possible only if no broadcast nor multicast is required.

As shown in Fig. 4.3, the software architecture contains three layers. The first layer is the hardware dependent layer *(Memory Management Unit)*, the second is responsible for the communication *(mail sending and buffer management)*, and the last layer handles the virtualization *(middleware protocol)*.

In [Baumann 2009] the authors propose an operating system specification dedicated to multicore architectures. This operating system should be distributed and

[3]National Taiwan University

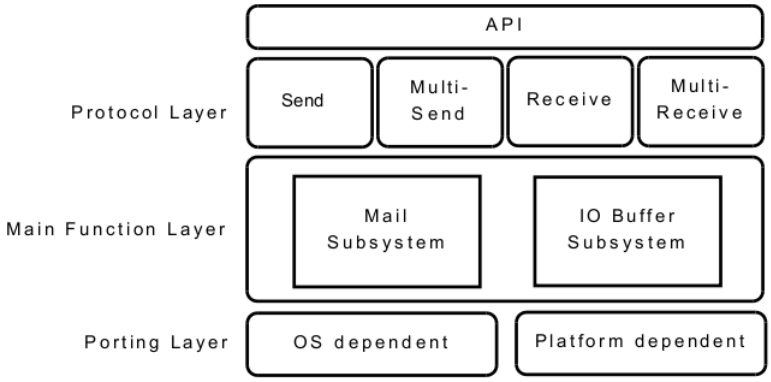


Figure 4.3: ICPC Service [Lin 2009]

provide explicit inter-core communication mechanisms. It also has to offer well-defined hardware abstraction layer and information about each core state should be replicated and not globally shared *(Fig. 4.4)*.

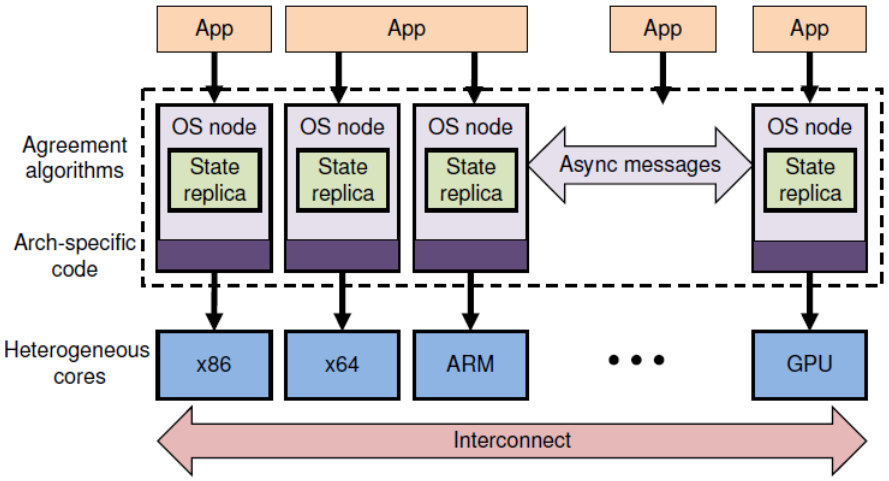


Figure 4.4: The multikernel model [Baumann 2009]

Facing the core heterogeneity, this OS would be flexible. Cache coherency management is not a necessity. Concerning inter-core communication, it would be realized through Message Passing mechanisms, what would make the system more modular and scalable.

At the kernel level, this operating system only manages the access to the hardware resources which correspond to the CPU drivers. Hence at the user level, each core would handle synchronization mechanisms, the memory management and the scheduling. In order to get more flexibility, each process in the application would

be represented by a dispatcher object present on each node the process can run on. It would be a kind of replica that we can active or deactivate regarding the core load.

Fos *(Factored operating system)* [Modzelewski 2009] is an operating system designed for manycore architecture. It has been defined to be scalable, easily extended and programmable, as well as able to perform automatic fault management. The authors seek to realize an operating system especially dedicated to applications which can take advantage of a cloud computing platform. In such a platform, number of available cores for one application is potentially unlimited. The main point is so to handle the scalability of these platforms.

As the operating system should be able to well balance the load of work on the different cores it controls, resources requests, that is to say computing power allocation for a given application is highly dynamic. Also, as the number of core is important, the operating system should detect if a core do not work anymore and in this case, modify the application deployment.

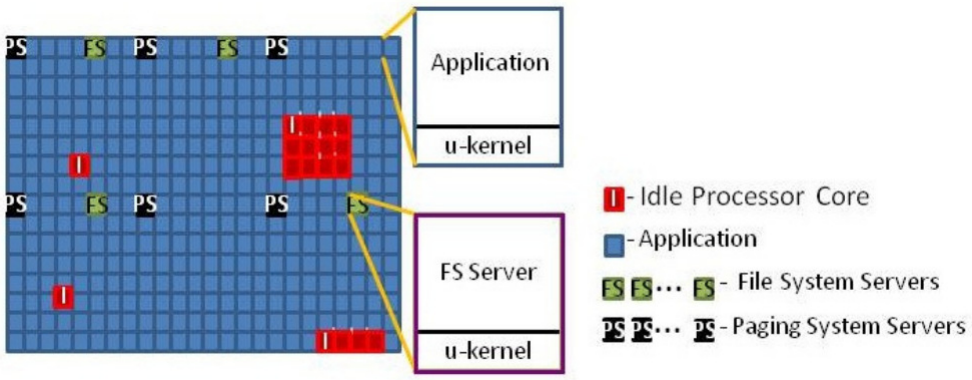


Figure 4.5: Factored OS [Modzelewski 2009]

Authors notice that, giving Linux as an example, the choice to keep a monolithic architecture and so to add locks on operating system shared structures in order to port it on multi-core platform, becomes more and more complex to do and hard to maintain. This is why they prefer to develop a new operating system based on the IaaS model *(Infrastructure as a Service)*, commonly used in the networked servers and virtualization fields *(Fig. 4.5)*. Fos relies on the following principles:

- to be adapted to a multicore architecture, the scheduling must be thought in two dimensions: time and space.
- for more safety, operating system servers must run on exclusive cores from the ones allocated for applications.
- operating system services are split into specific primitives, so each server can communicate through message passing with the other servers if ever it needs

a primitive implemented by these ones.

- servers whose primitives are complementary are grouped into "function specific fleet" in order to optimize their placement and reduce communication costs *(servers factorization)*.
- a server can be loaded or unloaded to increase or reduce resource use *(i.e. core use)*.
- resources used by an application have to be monitored in order to be able to efficiently manage fault appearance and to optimize platform resource use.
- in case of fault appearance on one of the servers, some are replicated to be used as substitute.
- the operating system includes a micro-kernel, a minimal kernel which have to be present on each core: it handles the hardware abstraction layer as well as application allocation and loading.
- a library called "OS Layer Server" permits to translate a system call performed by an application into a message towards the appropriate server. A special server called "Gateway Server" allows to go from one machine to another if necessary.

Synthesis:

Concerning this first aspect of the state of the art, we can say that due to the need of scalability, Inter-Process Communication via Message Passing seems to be the best choice when targeting heterogeneous multicore System-on-Chip. The operating system should follow an AMP strategy and be distributed over the platform. In addition, we should rely on a client-server mechanism to provide remote accesses to operating system services.

4.2.3 HRSoC middlewares

In addition to the multicore aspect, this dedicated operating system should be able to handle its heterogeneity, especially the presence of reconfigurable hardware component as processing elements.

In [Shiyanovskii 2009a], reconfiguration is handled by a software layer on top of a real-time operating system *(Fig. 4.6)*. This layer is called Adaptation Manager and is able to adapt in order to get a trade-off between the power consumption and the execution speed. It relies on a learning process which allows the manager to improve the latency to take a decision.

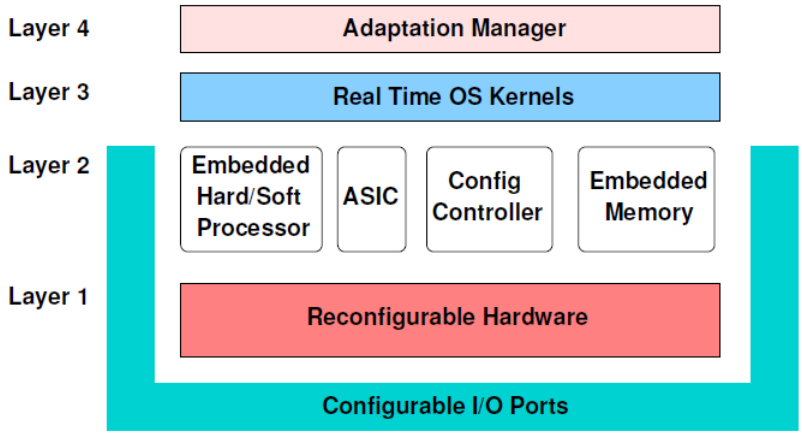


Figure 4.6: Self-reconfigurable platform [Shiyanovskii 2009a]

Actually, the reconfigurable platform is composed of tiles permitting to perform high-level functions regarding CLBs programming layer *(filters, FFT, ...)*. A priority-based scheduler is implemented to manage task execution. These tasks can have three different states: Inactive, Active or Reserved. The latter state is used to define a tile waiting for a task to arrive.

[Guerin 2009a] deals with the conception of an operating system dedicated to the heterogeneous multi-core systems-on-chip *(HMC-SoC)*. They start from the observation that the main approach which propose to have a standard processor interacting with hardware co-processors through some drivers is not adapted to complex system anymore. On the other hand, the specialized approach which consists to have one board support package *(BSP)* for each plateform coupled with a modular development is too generic to provide good performances. So they propose an intermediate approach based on components *(Fig. 4.7)*.

To achieve this, they need to define stable and generic interfaces as well as a clear segregation between hardware dependent components and hardware independent ones. The hardware abstraction layer is composed of 27 primitives responsible for:

- the endianness
- the multiprocessor configuration *(boot and cores identification)*
- the input / output conflicts
- the context handling
- the synchronization
- the traps

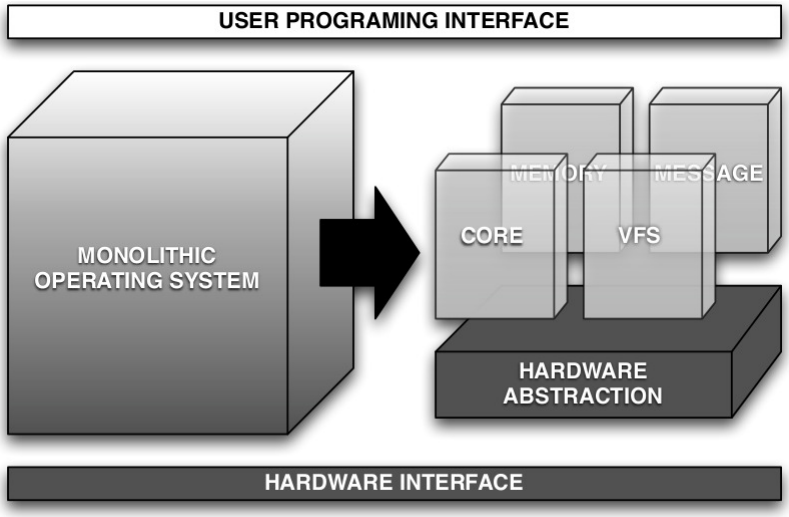


Figure 4.7: System framework overview [Guerin 2009a]

- the memory and the cache

The operating system which illustrates this approach is called DNA OS *(DNA is Not just Another Operating System)*. It is based on BeOS and offers thread management and scheduling services, a file system with or without MMU, dynamic memory management, Semaphore and Message Passing services as well as the ability to load each of these services dynamically inside the kernel.

In [Senouci 2006], the authors introduced a software architecture based on the Mutek kernel. The operating system is split in two parts: the HdS layer and the HiS layer. HdS stands for pour Hardware dependent Software and is a HAL managing the multiprocessor and the heterogeneous aspects of the cores *(boot, mutex synchronization and context switch)*. HiS stands for Hardware independent Software and is composed of the operating system, a middleware and the user layer.

They also propose a design flow allowing to specify this HdS layer. The implemented scheduler can manage SMP platform or one instance of it can be deployed on each core. In addition, one of the most important advantage of this OS is its low memory footprint.

Authors of [Matilainen 2011] propose an MCAPI implementation for the Systems-on-Chip. They choose MCAPI because it was though for inter-core communication, not inter-computer ones. An API is needed in order to develop efficiently complex portable applications. OpenMP [Board 2012] requires special support from the compiler which is not the case for MCAPI. Even if reduced MPI versions are available,

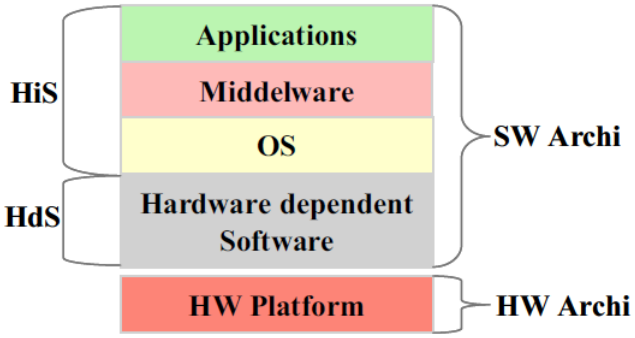


Figure 4.8: Hardware Dependant Software layer [Senouci 2006]

MPI requires more changes to source codes and CORBA [OMG 2006] is too heavy-weight. Regarding the implementation they did, it offers a lower memory footprint at the expense of less flexibility due to a limited number of calls.

MCAPI offers three types of communication: Message which is a basic message passing protocol, Packets which is a connected mode allowing to send or receive several messages in a row, and Scalar which permits to send or receive single fixed-size word *(Fig. 4.9)*.

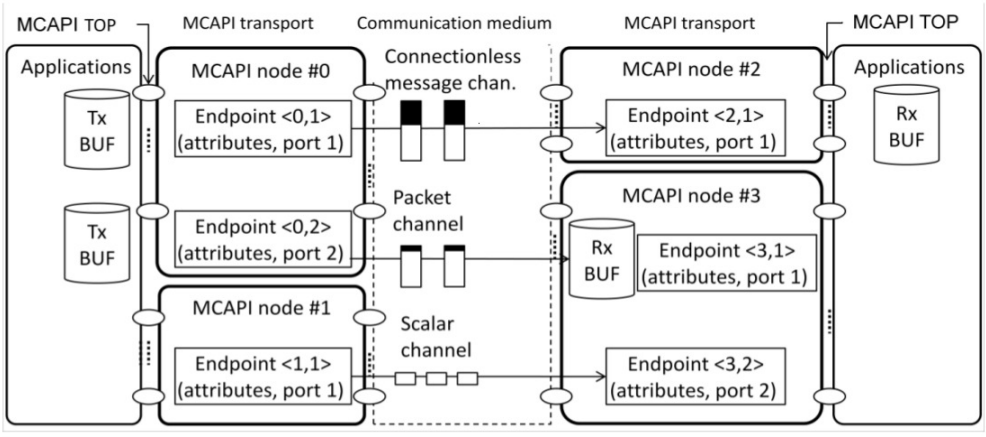


Figure 2. Example with 4 MCAPI nodes that are using the three available communication schemes. Communicating endpoints must have the same type, hence there are 6 of them.

Figure 4.9: MCAPI for MPSoC [Matilainen 2011]

The top layer implements MCAPI specified abstraction for user application and does only simple error checking for function calls. The underlying layer (Transport) implements the interface between the top layer and the HAL. Moreover, in this implementation, hardware accelerators are also seen as MCAPI nodes. It should be noticed that the node topology is static to make the implementation simpler.

In [Kamppi 2011] they designed an IDE[4] which allows to integrate some IP-XACT [541 2010] components together. It is open source and includes the generation of endpoints in order to be compliant with MCAPI.

Synthesis:

In order to manage the heterogeneity of a HRSoC platform, it is necessary to offer an additional layer on top of the operating system. This layer would help to provide a transparent access to the operating system services. Moreover, certain services are more likely to be efficiently implemented in hardware, so a hybrid operating system services should be proposed to the developer in order to improve the overall performances of the application and really take advantage of the heterogeneity of the platform.

4.2.4 Hybrid OS for HRSoC

The micro-kernel introduced in [Nordstrom 2005] is defined as a RTU[5]. The aim is to reduce the memory footprint of the kernel, taking advantage of the parallelism and enhance the kernel execution determinism.

To achieve it, some part of the operating system are implemented in hardware: the scheduler, the Semaphore and Flags services, an interrupt controller as well as timers. This RTU is based on the μC/OS-II kernel. When the paper has been published, all features have not been implemented yet. Nonetheless, we can notice for the ones which were implemented, that the gain is significant.

[Agron 2009b] asserts that a monolithic operating system is not adapted to multicore platforms anymore, parallelism causing important latencies for thread synchronization. Managing mutexes at the ISA layer *(Instruction Set Architecture)* using atomic instructions could be an efficient solution but is not really portable. Finally, remote procedure calls are too expensive in terms of time overhead.

So, the proposed idea is to port some features of a micro-kernel in hardware to light up the software part *(Fig. 4.10)*. The authors developed a Linux-based micro-kernel, flattened in order to simplify it, but always POSIX compliant. The Mutex service, the scheduler, variable conditions and thread management are the services chosen to be ported.

The scheduler manages tasks all over the cores and so acts like a SMP kernel. The advantage for a processor core is that it would be interrupted only when a preemption is necessary. The rest of the time, it quietly execute the thread the scheduler assigned it. This scheduler module is able to manage 128 priority levels and the Mutex IP provides two primitives: lock and unlock, requiring only one instruction to be performed *(atomicity)*.

[4] Integrated Development Environment
[5] Real-Time Unit

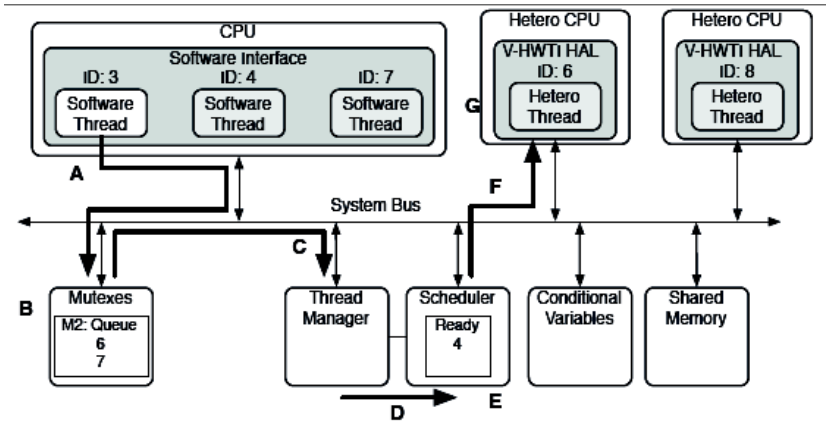


Figure 4.10: Hybrid Threads platform [Agron 2009b]

Finally, Götz et al. also proposes a hybrid solution in which the operating system services can be migrated during run-time from a software to a hardware implementation and reciprocally, depending on the application needs [Götz 2009]. An heuristic has been developed in order to optimize the resource use of each application that would be loaded on the platform.

4.2.5 Conclusion

The need of more and more computing power in the current embedded system pushes the designers to provides a new kind of system which are heterogeneous and more and more distributed. In order to adapt to the need of scalability of such a system, Inter-Process Communication became a cornerstone of the operating system. This communication should rely on a decentralized system. The Message Passing communication paradigm is well adapted to manage the communication of these heterogeneous multicore System-on-Chip. To go further, the operating system should also be decentralized. An Asymmetric Multiprocessor System would be the best choice to manage the numerous cores independently the one regarding the others. Additionally, the access to the operating system services would be based on the a client-server mechanism that offers a good scalability.

Regarding the heterogeneity aspect, the addition of a middleware layer would brought a significant abstraction to the HRSoC platforms. This layer would provides a transparent access to every available services in the platform. This flexibility would improve the platform partitioning allowing the developer to implement some services

in hardware and others in software, and also to choose on which processing units these services would be implemented. Such a partitioning would enhance the overall performances of the application as it would take into account of the advantages of each processing units.

This flexibility brought by distributed services will enhance the global performance gain allowing any thread to access to the most available efficient implementation of a given service. Moreover, basing the implementation of this additional layer on a widely supported standard would help the integration of the operating system in high-level design space exploration tools.

4.3 Specifications

Accordingly to the general information and the state of the art introduced previously, we first define the specifications of an ideal operating system for the HRSoC. It includes the constraints, the main objectives, and the chosen solutions regarding the state of the art. This specification will be a base to tackle the conception phase in which the details of the implementation of this operating system will be discussed *(See Section 4.4)*.

4.3.1 Objectives

Our goal is to abstract the heterogeneity of future multicore platforms, this in order to provide a fair access to any service for any thread in the system and also to define a homogeneous model of communication. We need distributed services to permit an optimized distribution regarding cores specificity and threads location.

Currently, emerging embedded systems tend to have a versatile general appearance, and are able to satisfy most of the final client's needs. This versatility, coupled with the increasing need of performances, modify their architectures into heterogeneous platforms. In order to satisfy these needs, they provide several processing units, each of these dedicated to a special task in the system, the whole forming a so-called multicore system, in which one core equals one processing unit.

In such a system, the developer can deploy his application onto general purpose processor, dedicated ones like DSP or ASIP *(Application Specific Instruction-set Processor)*, but also hardware accelerators running on reconfigurable chips especially used to perform recurrent and intensive processing, denoted as IP *(Intellectual Properties)* in FPGA devices.

In this section our objective is to define the structure and the characteristics of an operating system dedicated to this kind of system defined as multicore and heterogeneous.

4.3.2 Programming model

For the end-user, the application will be viewed as a homogeneous set of threads communicating through operating system services, wherever they are located *(Fig. 4.11)*.

On top of HRSoC platforms, the developer wishes in a first time to be able to validate his application without being dependent from its composition, for instance the number of cores or the type of these cores. This need of abstraction involves to add an intermediate layer between the hardware and the application. The deployment of the application and the operating system, meaning task placement and services distribution over the different cores, should be the most transparent as possible for the developer, and ideally handled by automatic tools.

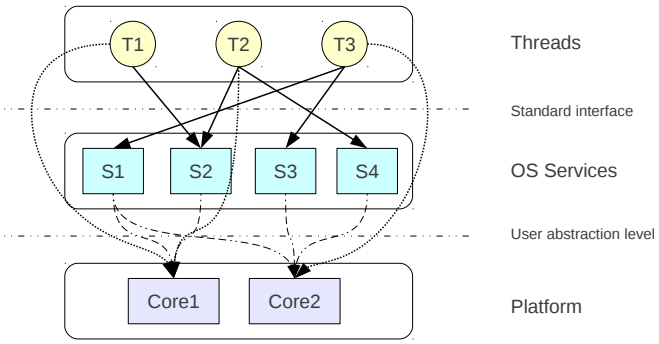


Figure 4.11: User point of view

The advantage is a simplification of the programming model thanks to a unique interface and an acceleration of the development process that can be reached through the automatic generation of code. The drawback is lowest performances and flexibility constraints among the genericity. The call to an operating system service should respect all or a part of an existing standard.

4.3.3 Memory constraints

We chose to implement a NUMA architecture, it means a distributed memory with Non-Uniform Memory Accesses. The operating system footprint should be consistent with the System-on-Chip capacity. Pragmatically, we set that the footprint of the kernel alone must be under 25 kB.

It is considered that the address space of the multicore system is a unified address space. All cores share the total system memory. Nevertheless each core will be able to have a private address space in which it will host its code and private data *(Fig. 4.12)*. The addition of a memory management unit *(MMU)* is not necessary.

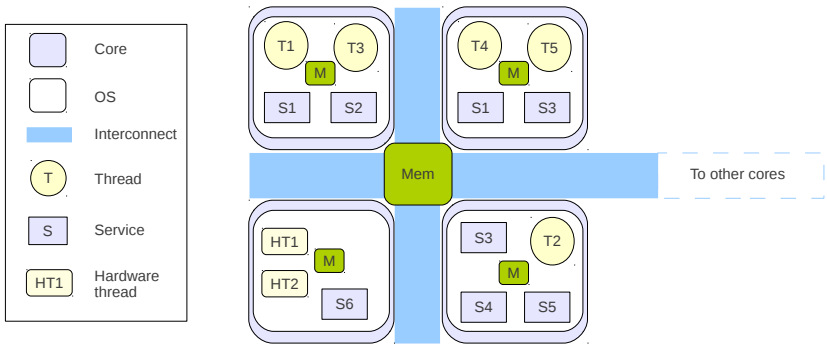


Figure 4.12: Platform memory architecture

4.3.4 Architecture

The specification of the operating system architecture includes the different modules which compose the multicore system, and the distributed servers.

4.3.4.1 Operating system architecture

The appearance of multicore system forced the operating system to be modular in order to take the most advantages of the parallel ability of the platform and also to optimize its memory footprint by sharing certain services among multiple cores. These services will therefore be seen as distributed services. For the sake of flexibility, we also wish a request to a service to be independent of its location. The service can be present on the same core than the thread which needs it, but can also be found on another core.

Each service performs a specific function. Some services will be performed by several other different services. These services will then need to communicate together. For example, a semaphore release by the service which manages the *Semaphore* will require it to inform the scheduler service that a thread can be unblocked.

Regarding the kernel architecture, the modularity constraint excludes the adoption of a monolithic kernel, more powerful but harder to port on a new architecture and very inflexible. Micro-kernel is more modular. For performances reasons, the inter-thread communication and above all the address spaces switch must be minimized. For this, the exo-kernels provide a solution even more flexible than the one provided by the micro-kernel. In most cases when designing embedded systems, an exo-kernel can be seen as a micro-kernel which the abstraction layer has been reduced to a minimum, namely the Hardware Abstraction Layer *(HAL)*.

The structure chosen for our kernel is closer to the exo-kernel than the micro-kernel as it exists today. The operating system servers are hosted in the user-space. For multicore platforms, the increase of the number of cores requires to increase the number of memories. For better management of the locality, a separation of this memory into multiple adjacent address spaces is required *(NUMA: several physical memories but only one unified address space)*. Therefore, the establishment of a

mechanism of MPU[6] which would combine flexibility and performance would be more feasible in an exo-kernel.

4.3.4.2 Services for multicore

An inter-core communication service should allow to send messages from one core to another. This service must be sufficiently generic to support different interconnect architectures *(Bus with shared memory, Network-on-Chip, ...)*. This module should also be able to manage the broadcast, which will be useful to synchronize several cores.

A localization function permitting to find existing services and resources on the platform must be present in each module to make itself independent of the threads, services and resources placement. The services are defined and statically distributed at compile-time. The resources are dynamically created at run-time. There are services and resources that are local, and others that are global, that is to say, available for the threads running on remote cores.

For more portability, resources will be accessible by a unique name, independent from the platform on which the application runs. A system of name resolution should offer the ability to identify a given resource using a user-defined name, statically or dynamically.

Regarding the inter-thread communication, the operating system should at least have a communication service. Tasks scheduling will always be done locally to each core. So there will be one scheduler service per core.

4.3.5 Portability

Constraints on the platform are its multicore aspect and its heterogeneity. It has an impact on the hardware abstraction layer which should take into account that one of the core should be appointed to initialize the platform and synchronize other cores. This core would play the role of supervisor. Additionally, the HAL should handle the heterogeneity of the cores and for instance the differences of endianness. To simplify the porting of the OS, the endianness must be switched *Big Endian* to *Little Endian* and vice versa. To simplify the process, a default endianness should be defined inside the platform.

4.4 Conception

The conception section describes how we choose to implement the specifications defined in the previous section. That means the operating system we choose to be implemented, how we manage the communication between cores and how we ensure the homogeneity of the programming model.

[6] Memory Protection Unit

4.4.1 Operating system architecture

As we made the choice of a distributed architecture, each core implements an instance of the operating system. The operating system is composed of a kernel and multiple servers. The role of the latter is to provide all application threads an access to the operating system services, and especially to the resources they manage. In this context, a resource is an instance of a Semaphore, of a Message Queue or of another service. It corresponds to the entity which is manipulated by the thread, using the service primitives. The distribution of the services on different cores may be uneven, depending on memory space, logic elements availability, core affinity or threads location.

4.4.1.1 Servers architecture

Servers are instantiated statically. During the creation of a resource, the search for a server capable of processing this work has been performed at compile time. When the server is local, the task wanting to run the service sends its message directly to this server locally. Otherwise, if it is remote, the task sends a message to the remote node on which it stands.

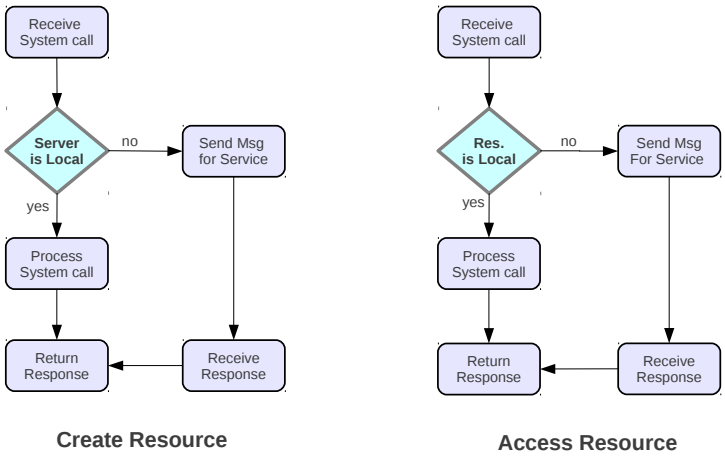


Figure 4.13: Syscall Procedure

Resources are instantiated dynamically. When handling an existing resource, a thread has to find where the resource is located, then send a message to the server owning this resource. If the server is on another core than the thread, a message must be sent to the core in order this one to transmit it to its local server *(Fig. 4.13)*.

Regarding its architecture, a server is a module that implements the mechanisms for handling a certain type of resource. These mechanisms, when the service is implemented locally, are a mechanism of location and a mechanism of resource management *(Fig. 4.14)*. These two mechanisms are optional. If not implemented,

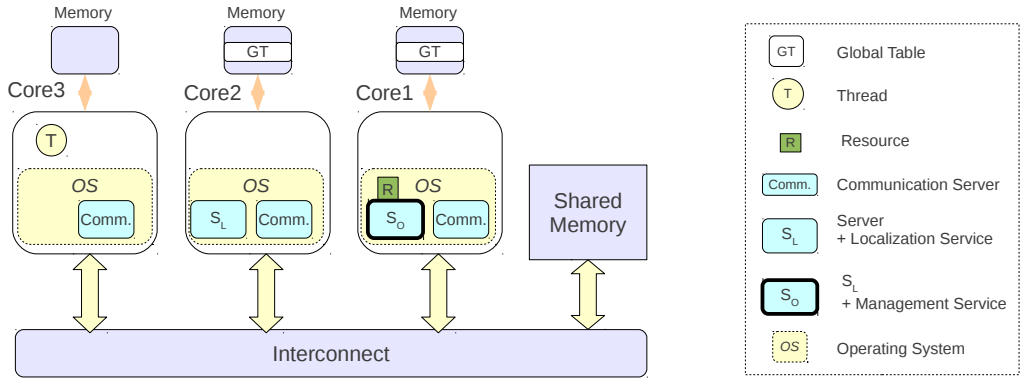


Figure 4.14: Server types

the server is restricted to the communication service allowing to send and receive messages, and so to access to remote servers or resources using Inter-Thread Communication *(Fig. 4.15)*.

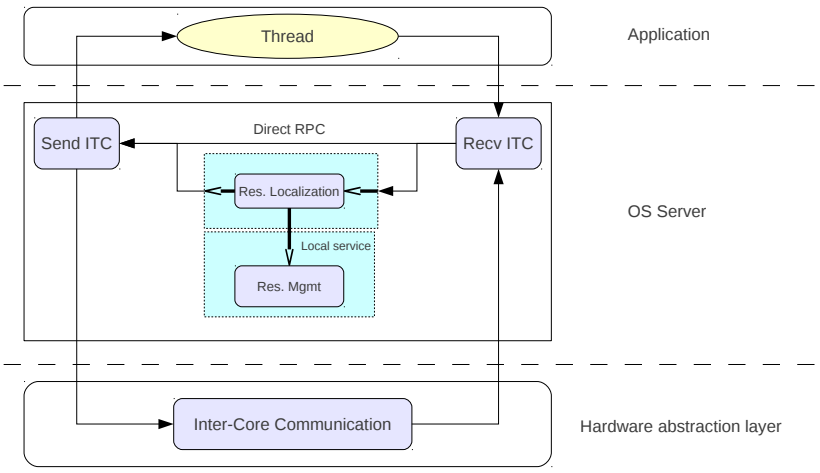


Figure 4.15: OS Server Architecture

4.4.1.2 Kernel architecture

The kernel on each core will be implemented as an exo-kernel and so will provide a minimal set of features. It must be composed at the functional level of: a HAL[7] which gives access to a timer tick for the operating system, an interrupt module, a bootloader, a thread management server, and finally an inter-core communication server allowing access to remote servers and resources.

[7] Hardware Abstraction Layer

4.4.1.3 Communication architecture

A user thread accesses the services of the operating system through a specific API. Several standard APIs exist. Whatever the chosen standard, it is necessary to homogenize the system calls for all types of threads that contain the application, which could be either software or hardware. For scalability reasons, communication between a server and a thread is made only by message passing. The message must contain all information necessary to enable the server to execute the query. This stateless protocol is intended to limit the number of transactions between a thread and a server when performing a system call. Similarly, the communication between two servers is done by encapsulating the message in a routing header specific to the physical communication medium *(Fig. 4.16)*.

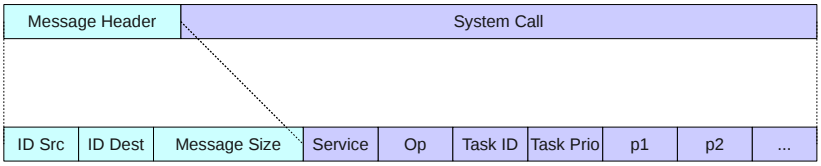


Figure 4.16: Message Template

All messages exchanged in the system have the same format. They consist of two segments:

- system data *(System Call)*: this segment indicates the service to which it corresponds, the operation which is requested to do on this resource, as well as the identifier and the priority of the calling thread if any. The following data are the parameters of the requested operation
- the header needed to route the message through the interconnect in the case of a communication between two cores

4.4.2 Platform architecture

4.4.2.1 Hardware architecture

To illustrate the different mechanisms that we need to implement, we rely on the platform described in Figure 4.17. It includes three cores: *Core0*, *Core1*, and *Core2*. *Core0* supervise the entire system and is responsible for initializing the platform and starting the other cores. Each core has access to a private memory, a shared memory and can communicate with other cores through an interconnect.

Core0 and *Core1* implement both locally a Semaphore service. *Core2* does not implement it. The application is composed of two threads, *T1* and *T2*, respectively

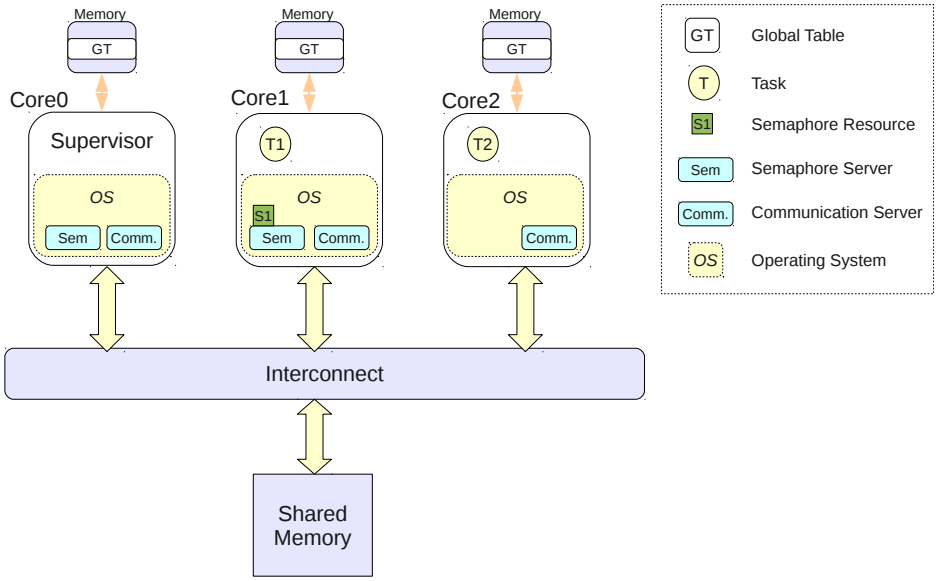


Figure 4.17: Study Case Platform

present on *Core1* and *Core2*. At T_0, we consider that the supervisor has configured every nodes and they are ready to execute the threads they implement.

Each core has a local scheduler. The resources created by the servers can be local or global. In the case of global resources, information on this resource are filled in a local table, reflected in the private memory of each core *(GT = Global Table)*. This table allows each core to locate any global resource created in the platform.

4.4.2.2 Study case

Scenario

The application scenario is as follows: *T1* creates the global Semaphore *S1* then releases it. *S1* is initialized to 0 *(ie. there are no resources available)*.

```
1 T1(){
2   create_semaphore("S1", GLOBAL);
3   release_semaphore("S1");
4 }
```

T2 starts waiting for Semaphore S1.

```
1 T2(){
2   create_semaphore("S2", GLOBAL);
3   take_semaphore("S1");
4 }
```

Scenario steps

T1 creates the global semaphore *S1*. *Core1* implements a Semaphore service so the request is processed locally. When *S1* is created, *Core1* warns the other cores that a new global Semaphore has been created. To do so, it sends a message to each core.

When a core received this message, it updates its global table *GT*. The content of the global table is detailed in *Table 4.1*. Depending on the case chosen for the implementation of Semaphore service, some fields would be left blank, for example the Attribute and Pointer values are not neceplacedssary when the resource is distant.

For its part, *T2* wants to create a new semaphore. Since it does not implement the service, it must call this service on *Core1*. To consider the establishment of remote services calls, it is necessary to add a proxy mechanism, or replica, which will emulate the presence of thread on the remote core *(Fig. 4.18)*. For reasons of space and memory latency, this proxy must contain the minimum information necessary to be managed by the scheduler.

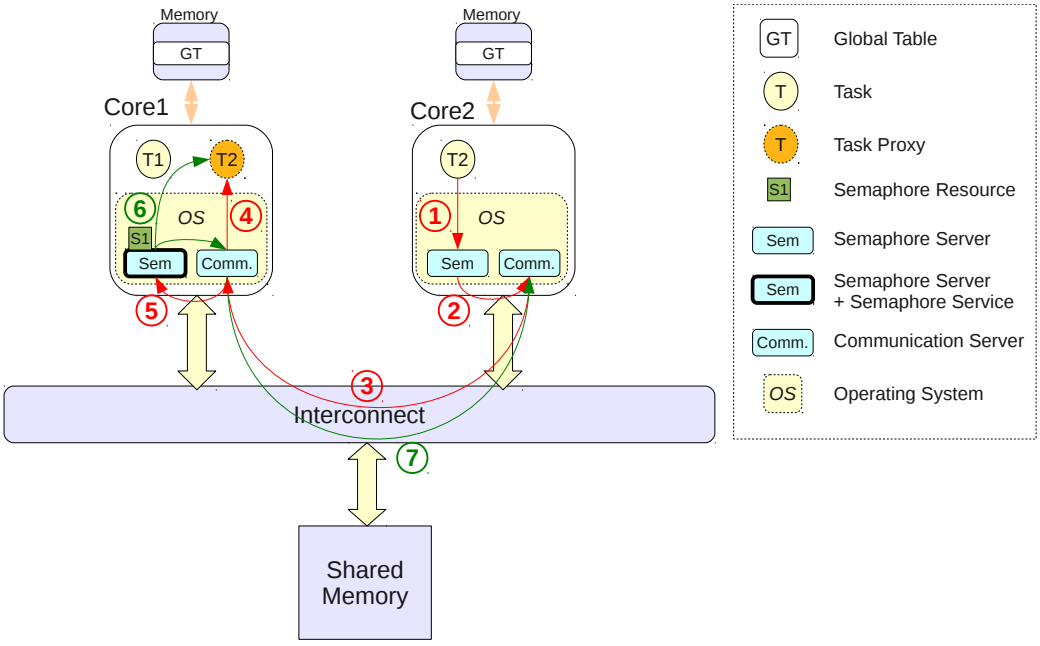


Figure 4.18: Distant system call

The creation of Semaphore *S2* unfolds as follow *(Fig. 4.18)*:

(1) *T2*, located on *Core2*, performs a system call which is translated as a message to its local Semaphore server.

(2) as the local server does not implement the service, the call is directly routed to the Communication server. It therefore starts an inter-core communication to

have the *Core1* perform the request.

(3) the Communication server sends the message across the interconnect.

(4) the communication server of *Core1* receives the message. It then creates a replica of *T2*.

(5) once the replica is created, it forwards the call to the server so that Semaphore performs the service requested by *T2*.

(6) when the service is performed, the proxy is destroyed and the feedback information is transmitted to the inter-core communication service.

(7) the feedback information is sent to *Core2* for transmission to *T2*.

Then *T2* uses the resource *S1*. It does not implement the service, thus the service implementation at compile time is reduced to a direct call of the remote service, disregarding whether the resource is global or local since it is necessarily global. It therefore locates *S1* through its global table.

It is considered that the resource *S1* is managed locally by Core1 and directly inaccessible by the other nodes. All requests for an operation on *S1* must be done by *Core1*. In the global table, the value associated with each resource is the core identifier on which it is located.

Resource	**Core ID**	**Status**	*Attributes*	*Pointer*
S1	1	Created	Shared	0x90000150

Table 4.1: Resources table example

In this case, *Core2* sends a message to *Core1* specifying the identifier of the Semaphore and the request *(Semaphore locking)*. The message is received by *Core1*, via a thread dedicated to this task. The request of *T2* is performed on *Core1* and a proxy of *T2* is placed in the waiting queue of *S1*.

The advantages are that this solution is scalable because all communications are done by message passing and that there is no conflict about the ownership as the resource is still managed by the creator of the resource. On the other hand, the drawback consists in the fact that all operations on a resource are centralized on creator's location.

When *T1* releases the semaphore, *T2* is the highest priority thread waiting and therefore takes the Semaphore. The information is returned to *Core2* in another message. *T2* is then released and becomes ready to run.

Possible scenarios

When a thread uses a service of the operating system, there are three possible scenarios:

1) the **server** is implemented **locally**, and the **resource** is **local**. In this case, after locating the resource, the thread calls directly the local server and the resource is handled directly by the service *(Fig. 4.19 and 4.20)*.

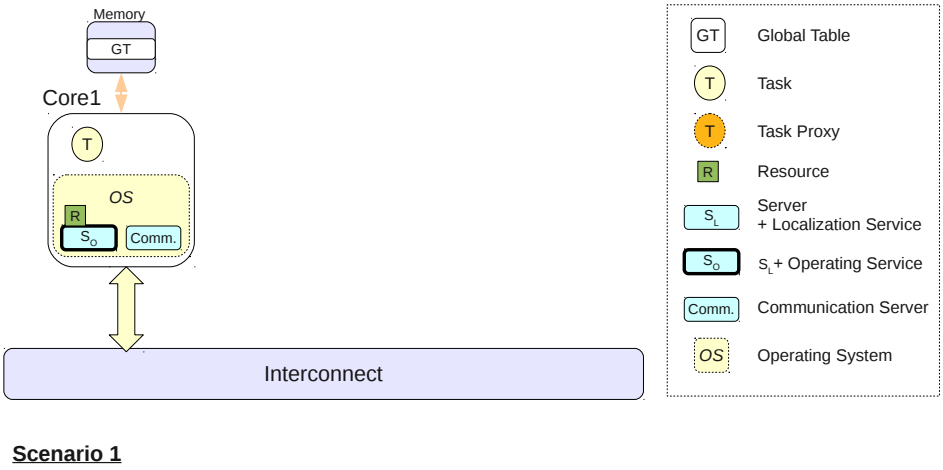


Figure 4.19: Scenario 1 platform

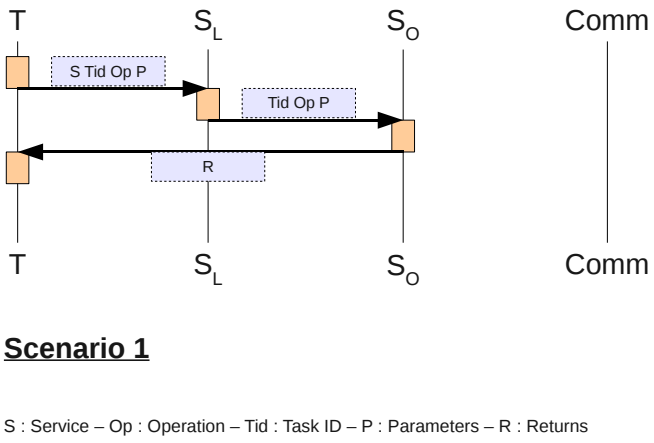


Figure 4.20: Scenario 1 datagram

2) The **server** is implemented **locally**, and the **resource** is **remote**.

   The **server** is present **locally** but does **not** have the **resource**, it must first locate it. Once located, it is responsible for sending a message to the core which possesses it for the latter to process the request in its place *(Fig. 4.21 and 4.22)*.

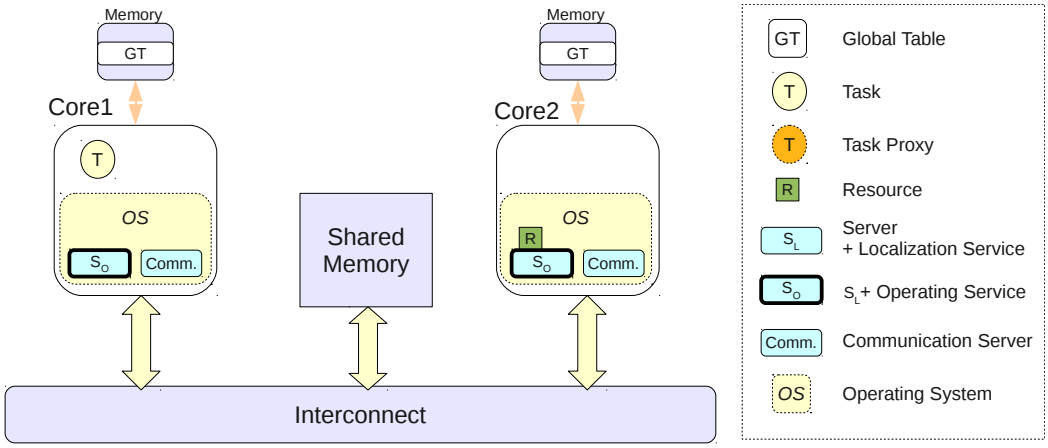


Figure 4.21: Scenario 2 platform

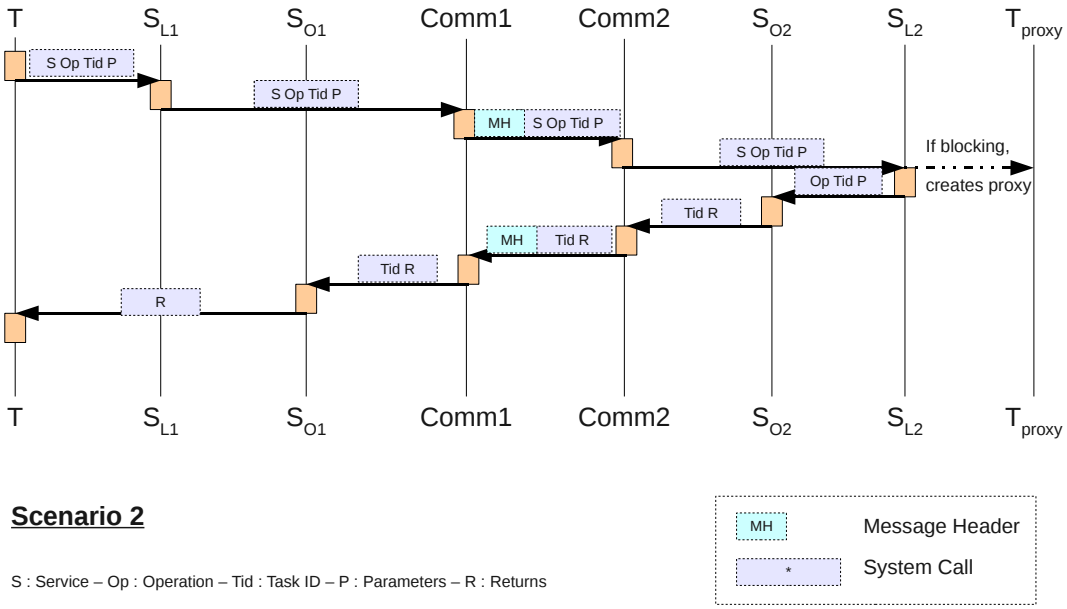


Figure 4.22: Scenario 2 datagram

3) The **server** is **not** implemented **locally**, and the **resource** is **remote**.

A **service not** implemented **locally** is declined into two different versions:

a) The **server** implements a **localization** mechanism: the **server can not handle resources**, but it is **able to locate the resource**. Once located, it sends a message to the core which own the resource *(Fig. 4.23 and 4.24)*.

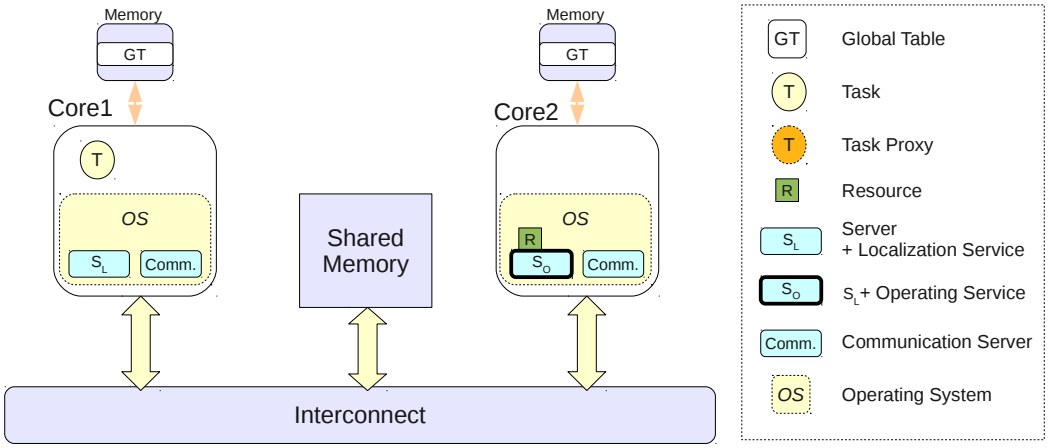


Figure 4.23: Scenario 3a platform

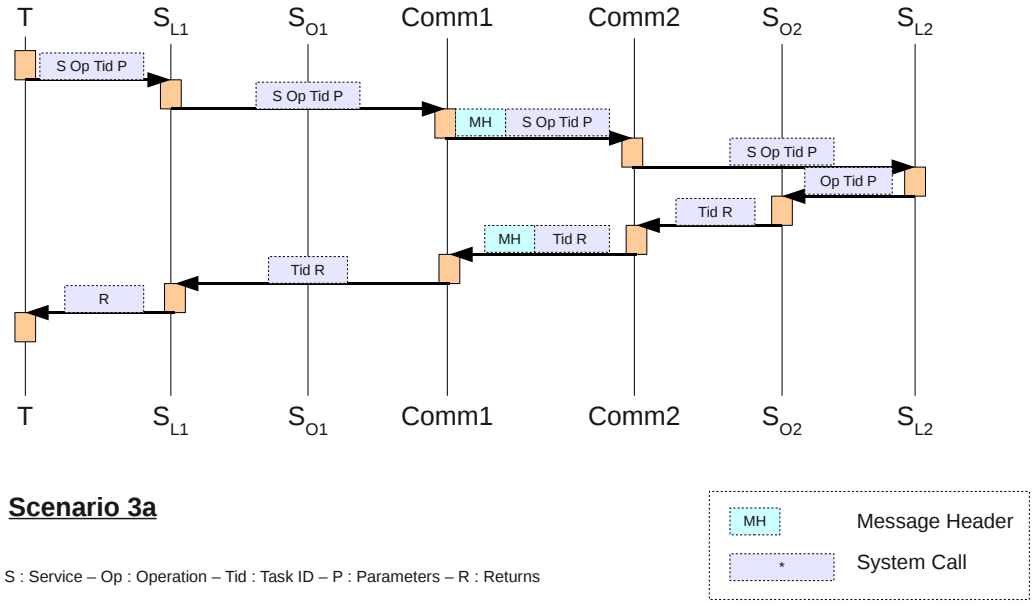


Figure 4.24: Scenario 3a datagram

b) The **server** is **not implemented** *(Fig. 4.25)*. The call for this service has resulted in sending a message to another core which is known to own the service. Then that other core has the resource, or by extension, is able to locate and deliver the message to core which effectively owns it *(Fig. 4.26)*.

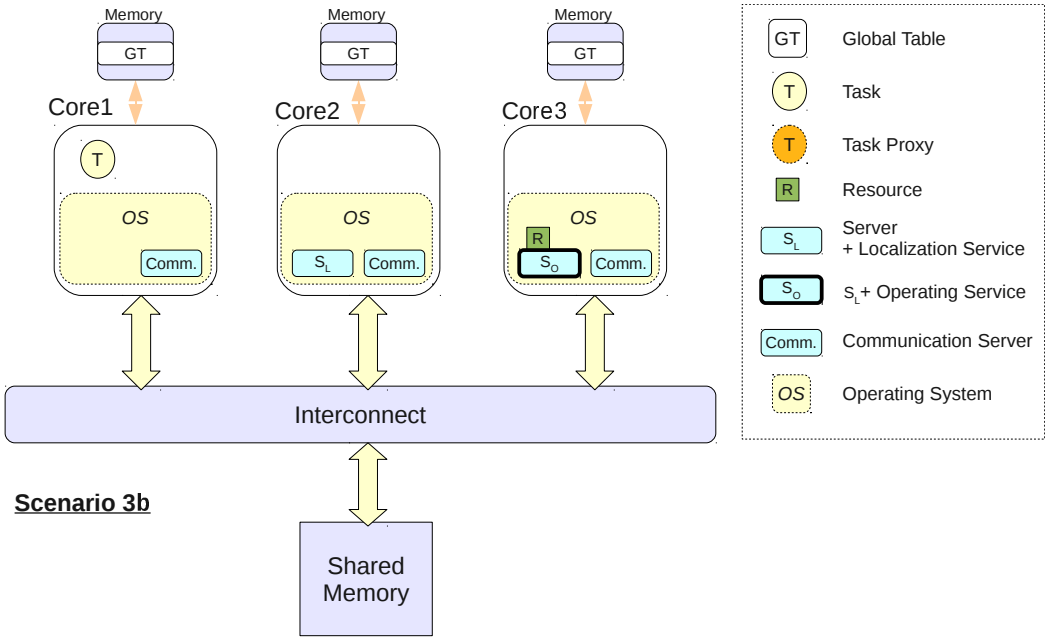


Figure 4.25: Scenario 3b platform

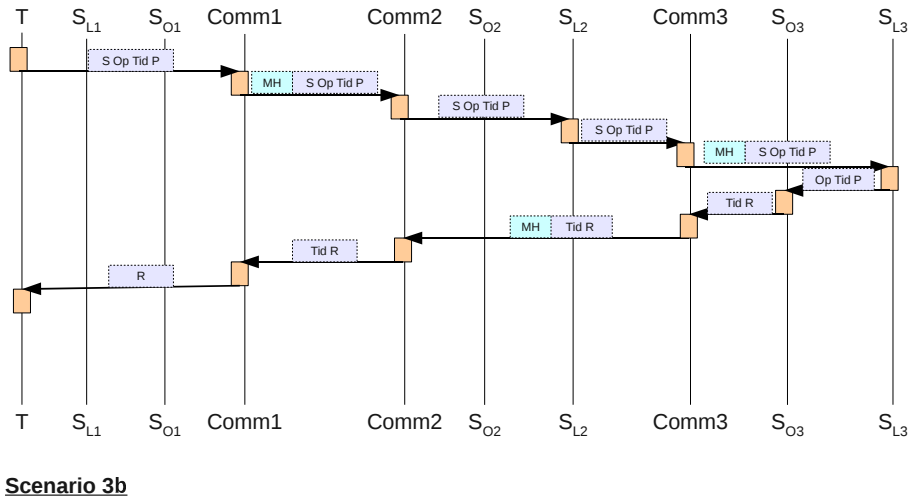


Figure 4.26: Scenario 3b datagram

4.4.3 Multicore layer

The abstraction provided by the software architecture has to be integrated in an operating system. We have to keep up the existing structure of this operating system, especially the services it provides. In a common operating system, a thread accesses the operating system services via classical system calls *(ie. direct connection to the called primitive)*. To abstract remote accesses to an operating system resource,

an additional layer must be added. This layer has three objectives:

i) to make the difference at compile time, between a request for creating remote or local resources

ii) to translate calls to the operating system primitives into messages understandable by the servers

iii) to manage the heterogeneity of the platform: differentiation of how to manage a software thread with how to process with a hardware thread

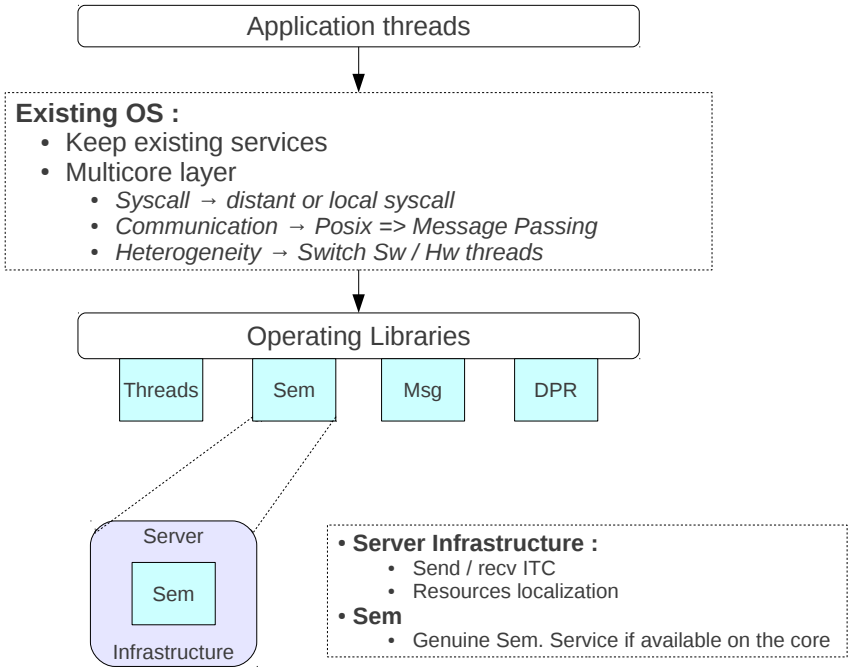


Figure 4.27: Operating system architecture

The implementation of this multicore layer requires us to identify the existing mechanisms permitting to the different nodes of the system to communicate. These mechanisms must be adapted or modified to allow sending a message from a core to another and to ensure the transfer of all the necessary information. Finally, the operating system should implement the following modules:

- a module that implements a service of **message passing**
- one or several modules that allows to **manage the resources** available on the platform *(e.g. a Semaphore service)*
- a module enabling to abstract the use of the **partial and dynamic reconfiguration**

4.5 Implementation

4.5.1 Modular operating system: MutekH

To validate the choices and concepts to be implemented in order to realize this operating system, we will use an existing operating system: *MutekH* [LIP6 2012]. This operating system was chosen because it is a multicore heterogeneous operating system, open source and currently still maintained by the LIP6 laboratory *(www.mutekh.org)*.

4.5.1.1 Main features

The following table lists the features of this operating system and compares them with what is expected of our *"ideal"* operating system *(HSoC OS)*.

Scheduling	**HSoC OS**	**MutekH**
Type	Preemptive	Preemptive
Criteria	priority	round robin
Max. number of task.	>= 16	unlimited

Thread service	**HSoC OS**	**MutekH**
Create	Yes	Yes
Delete	Yes	Yes
Suspend	Yes	Yes
Resume	Yes	Handled by the scheduler

Mutex service	**HSoC OS**	**MutekH**
Create	Yes	Yes
Delete	Yes	Yes
Blocking take	Yes	Yes
Non-blocking take	Yes	Yes
Release	Yes	Yes
Priority inheritance	Yes	unknown
Deadlocks management	Optional	No, non-blocking take possible

Semaphore service	**HSoC OS**	**MutekH**
Create	Yes	Yes
Delete	Yes	Yes
Blocking take	Yes	Yes
Non-blocking take	Yes	Yes
Release	Yes	Yes

Message Passing service	**HSoC OS**	**MutekH**
Blocking send	Yes	No
Blocking receipt	Yes	No
Non-blocking send	Yes	No
Non-blocking receipt	Yes	No

Memory allocation service	**HSoC OS**	**MutekH**
Fixed allocation	Yes	Yes
Dynamic allocation	No	Yes

Remote communication	**OS HSoC OS**	**MutekH**
Resource creation	Yes	No
Resource destruction	Yes	No
Resource manipulation	Yes	No

Debug - Monitoring	**HSoC OS**	**MutekH**
Support GDB	Optional but recommended	OK
Statistics	Optional	Yes
Hooks	Optional	Yes

Hardware Thread service	**HSoC OS**	**MutekH**
Create	Yes	No
Delete	Yes	No
Suspend	Yes	No
Resume	Yes	No

Hardware Threads scheduling	**HSoC OS**	**MutekH**
Preemption	Save and restoration through readback	No
Relocation	Yes, on homogeneous areas	No

Multicore support	HSoC OS	MutekH
Bootloader management	Supervisor processor	Supervisor processor
Task migration	Optional	Yes, pointer to the code in shared memory

Features	HSoC OS	MutekH
Kernel footprint	< 25 ko	–
Memory safety	Memory Protection Unit	Memory Management Unit
Microblaze Port	Yes	Partially done (functional)
Abstraction API	Industrial standard	POSIX standard
Space address	Unified for every cores	Shared memory
Modularity	Modular OS services	OK

4.5.1.2 Services structure

Currently, MutekH offers modular services to be implemented on the target platform. Being an exo-kernel, additional services are defined as libraries *(Fig. 4.28)*. These libraries are separated into two categories: OS Interface Libraries whose APIs are provided by the user and Services Libraries whose APIs are provided by the operating system.

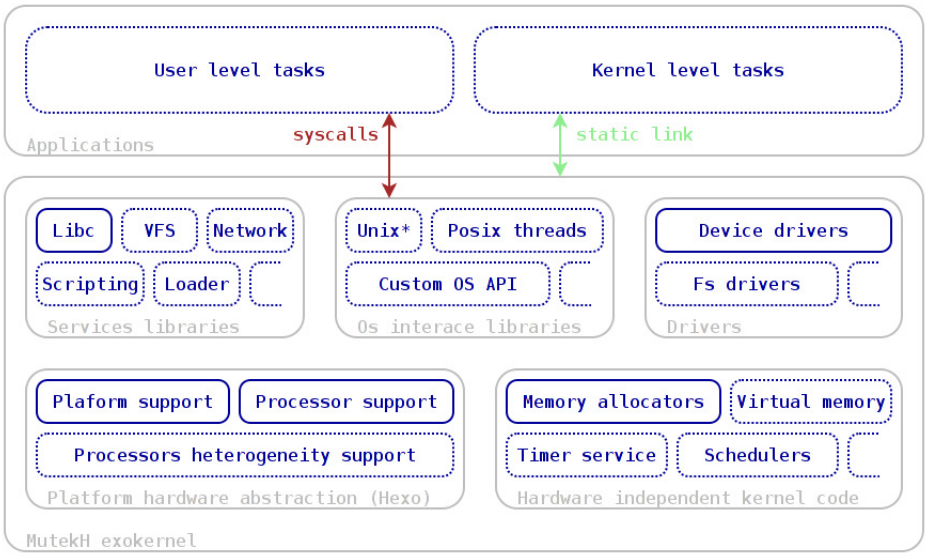


Figure 4.28: MutekH global view

The core of MutekH, Hexo *(Hardware independent kernel code)*, provides the following services:

- memory allocators
- memory regions
- page allocator
- scheduler
- timer
- semaphore

In order to satisfy the conception requirements described in the previous section, we need to add a new library which will be used as an multicore resource manager. This library has to handle the local and remote accesses to every resources of the platform providing a common API for every servers of the platform. In our case, we rely on the MRAPI specification provided by the MCA *(Multi-Core Association [Association 2012])*. This specification which offers an API to access to global services is detailed in the next section.

4.5.2 MRAPI Specification

4.5.2.1 MRAPI definition

MRAPI *(Multi-Core Resource Management API)* is a specification which aims to offer a standard API defining basic synchronization mechanisms, memory accesses and system metadata. Synchronization mechanisms includes Mutexes, Semaphores and pairs of Reader/Writer locks. Accessed memories can be shared or remote, whereas system metadata addresses the collect of hardware informations.

Their approach consists in suppressing the dependency of the existing standard with the SMP architecture and provide an API which could be easily implemented on a distributed operating system containing heterogeneous cores and shared resources. The advantage of using a standard API is the portability as a developer will be able to locate the non-portable functionalities.

MRAPI shares the same concepts as those found in MCAPI. It is orthogonal to this specification and the two are inter-operable. In these specifications, a system is composed of:

- domains: a domain is a system component which includes a certain number of nodes
- nodes: a node is an independent thread of control. It may be a process, a thread, a hardware accelerator or an operating system instance

By default, most resources are shared between different domains of the system. For efficiency reasons, it is possible to disable this by setting the attribute sharing MRAPI_DOMAIN_SHARED to MRAPI_FALSE when creating the resource.

4.5.2.2 MRAPI Mutexes

A mutex can be declared as a global resource by specifying the process-shared attribute. The Mutex is based on POSIX mutexes. They must support the detection of deadlocks and in this sense are similar to the implementation of mutex type PTRHEAD_MUTEX_ERRORCHECK. The sharing of a mutex between multiple processes is not always possible. This is implementation dependent. In particular in the case of the use of a fork.

They also support recursion but this is not the default case. For each lock, a unique key is returned and is used to check the order of calling Unlock primitives for the same mutex.

Regarding other features, the priority inheritance mechanisms are not guaranteed until the specification of threads in MTAPI[8] is not clearly defined. The operations on mutexes are all blocking and by default, a mutex is visible to all processes and tasks. The primitives defined by the API are equivalent to the following POSIX primitives:

- pthread_mutex_init (mcapi_mutex_init)
- pthread_mutex_destroy (mcapi_mutex_destroy)
- pthread_mutex_lock (mcapi_mutex_lock)
- pthread_mutex_trylock (mcapi_mutex_trylock)
- pthread_mutex_unlock (mcapi_mutex_unlock)

Mutexes have attributes. These attributes must be defined before creating the mutex and can not be changed later.

4.5.2.3 MRAPI Semaphores

Semaphores are also based on the POSIX standard. All operations are blocking and by default a Semaphore is visible to any process or task. This service also provides primitives for notifying deadlocks.

However, MRAPI only supports named Semaphores and the XSI interface *(X/Open System Interfaces Extension)* is not supported. Moreover, like Mutexes, the mechanisms to fight against priority inversion are not guaranteed as MTAPI is not completed.

[8] Multicore Task management API

4.5.2.4 MRAPI Reader/Writer Locks

The Reader / Writer Locks can handle multiple concurrent accesses to read from a memory, or exclusive access for writing. To ensure fairness, MRAPI must implement a mechanism for serializing queries so that no new read request is accepted from the moment where a write request is pending. MRAPI Reader / Writer Locks are similar to POSIX R/W Locks, but as MRAPI provides additional functionalities, MRAPI locks implementation is more flexible and for instance, locks can be shared by all nodes as well as by only a group of nodes.

4.5.2.5 MRAPI Memories

MRAPI supports two types of memory: shared memories and remote memories. MRAPI shared memories are similar to POSIX shared memory, except that they extend their functionality to several operating system, against one for POSIX. MRAPI supports the heterogeneous elements of execution and ensures consistency of shared memory regardless of the operating systems or the types of cores used to compose the platform.

Remote memories relate memories accessible only through mechanisms external to the processor, that is to say other types of instructions than simple load and store. There are no constraints on how to access these memories, however, it is preferable to provide an implementation in which the sending of data and calculation of data can be done in parallel. That is to implement mechanisms of non-blocking communication with the memory *(read and write)*.

In addition, flush and sync primitives are provided to support any cache management, and access of scatter / gather type. There are two types of access to a remote memory, all to be uniform:

- access with strict semantics: the type of access must be specified upon the creation of the buffer (e.g. DMA, software cache, ...)

- access without semantics: the type of access specified upon the creation is set to MRAPI_RMEM_ATYPE_ANY. The actual type of access is given only when accessing the buffer.

The use of pointers is still allowed but limited to access to local memory. Remote access must always be done making a copy and must use the MRAPI primitives. The implementation must still provide at least the default type of access[9] which must follow the strict semantics.

[9] MRAPI_RMEM_ATYPE_DEFAULT

4.5.2.6 MRAPI Metadata

Metadata provides access to information about the hardware platform. You can access this information by using the primitive mrapi_resources_get() which returns the information as a tree. Each node of the tree represents a system resource and has attributes giving additional information about the resource. The information contained in the tree can be filtered using the input parameter subsystem_filter. The implementation of these filters depends on the implementation.

4.5.2.7 MRAPI Attributes

The attributes were defined to allow an extension of the API. It is possible to define additional attributes specific to its own implementation. In order to make the API as portable as possible while keeping a flexible implementation, the attributes are maintained in a data structure opaque, non-visible to the user. Each resource is associated with a data structure and must have certain attributes and default value. These values are defined in the specifications. Three primitives are used to manipulate the attributes:

- mrap_<resource>_init_attributes()
- mrap_<resource>_set_attribute: to be repeated for each attribute to set
- mrap_<resource>_create(): takes as parameter attributes
- mrap_<resource>_get_attribute()

Note: Once the resource is created, its attributes should not change. For a resources management like remote resources, an additional layer should be implemented. In our case, this additional layer is brought by the servers architecture.

4.5.2.8 Non-blocking calls

There is three types of primitives:

- blocking primitives
- non-blocking primitives: it means primitives containing the word "_i" at the end of their name, indicating that it returns immediately
- and "single-attempts blocking" primitives: namely, primitives including the word "try" in their name

Remote memories are the only one to support non-blocking calls. In this case, the primitive can return the focus to the user before the completion of the operation. To check this completion, the API provides the following primitives:

- mrapi_test() : test the operation test without being blocked
- mrapi_wait() : wait for the completion of an operation or until a timeout
- mrapi_wait_any() : wait for the completion of one of the running operation given as parameters, or until a timeout
- mrapi_cancel() : cancel an operation

4.5.3 Hardware architecture

4.5.3.1 Homogeneous NoC-based platform

In order to validate the communication mechanisms between two processors, we first design a homogeneous platform. In this platform, each processor is master on its own bus and can access to the NoC resources through a bridge. A Bram memory is used to test read and write mechanisms on the NoC.

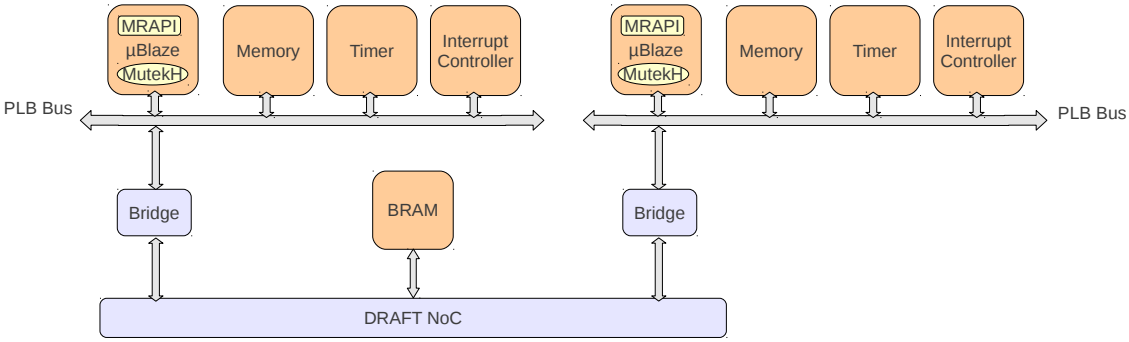


Figure 4.29: Homogeneous NoC-based Platform

4.5.3.2 Development environment

The MutekH operating system [LIP6 2012] has been ported on the Microblaze processor. To enhance its programmability, we developed an Eclipse plug-in which can be integrated into the Software Development Kit *(SDK)* provided by Xilinx to program the Microblaze. This plugin allows to create a new project to deploy MutekH on a Microblaze processor, choosing the libraries to include into the kernel and the memory mapping of the application.

4.5.3.3 Heterogeneous NoC-based platform

The heterogeneous platform we realized extends the homogeneous platform adding Hardware Thread instances *(HT)*. These tasks perform system calls through the Reconfigurable Zone *(RZ)* bus which is a bus dedicated to the hardware tasks in the FOSFOR project. Messages on the bus are recovered by a hardware communication

server responsible of translating requests into messages routed by the NoC towards a processor. Experiments on this platform will be described in the Chapter 5.

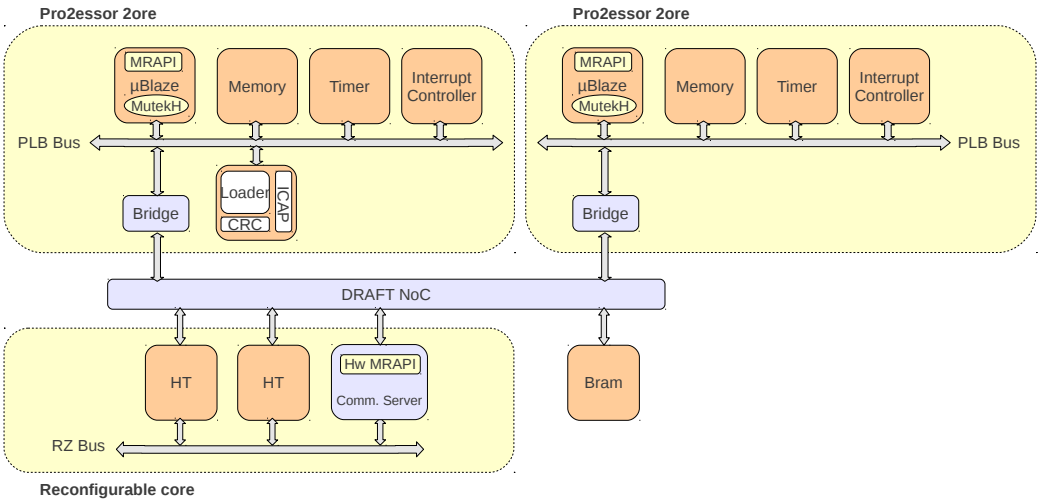


Figure 4.30: Heterogeneous NoC-based Platform

4.5.4 Domain definition

The platform is considered to be a NoC-based design and each core, processors cores or reconfigurable cores, have a unique identifier on the network-on-chip.

We could have defined a domain as a Network-on-Chip, however this would have led us to define a core as a node. This situation would be problematic because as the node identifier should also be unique, a core would have to ensure that the number it chose is not already used by the other cores connected to the network when creating a new node. Another solution would be to request to a core considered as a supervisor, to generate a number for it, what could also be a bottleneck.

For these reasons, we decide that a domain is defined by a core. As every core are already connected and defined on the NoC by a unique identifier, the domain number is derived from this identifier. So, to attribute an identification number to a new node, a core performs it independently from the others cores managing a local table.

This solution considers that in the platform, only the hardware thread can be reconfigured. In this implementation, we do not take into account the possibility to reconfigure a dynamic partition instantiating a processor core as defined in the Figure 4.30, but only instantiating a Hardware Thread.

4.5.5 Node definition

As regarding the MRAPI specification a node should be an independent thread of control. In MutekH, we defined that a software node is implemented as a POSIX thread. This choice has been made because a node is considered to be mapped anywhere on the network regarding application needs, so a node number is dynamically generated by a core when initializing the node.

On the hardware side, a hardware thread running in a reconfigurable partition is consider as a node, and like the software operating system, the Hw MRAPI module is responsible for generating the node identifier of each hardware threads.

4.5.6 MRAPI types

In the Multi-Core Association *(MCA)* implementation, used types are defined in the file *mca.h*. Defined types are prefixed by "mca_". Symbolic constants are then defined in order to homogenize the different implementations under generic types prefixed by "mrapi_".

In our case, the implementation is done in the *mrapi_impl_spec.h* file. We associate "mrapi_" types to "mmh_" types *(MRAPI MutekH)* defined in the file "mmh.h", which are themselves linked to already defined types in the MutekH kernel *(hexo/types/h) (Fig. 4.31)*.

The implementation is done to be deployed on a 32-bit wide architecture. Regarding primitives implementation, it exists two levels of files: mrapi.h and mrapi.c including API primitives as defined in the specification which only check specified errors messages, and mrapi_impl_spec.h and mrapi_impl_spec.c including the effective implemented API primitives which content is specific to the targeted operating system and platform.

4.5.7 Resources system calls

4.5.7.1 Principle

A thread wanting to access a system resource, which could be local or distant, can do it in a transparent way using the MRAPI primitives. A call to one of these primitives when concerning a system resource, is translated by a call to a flexible server.

A flexible server is an operating system service which could be implemented in three ways:

- with a **minimal** service **unable to localize a resource** but **able to transmit a request** to access to remote services to another core
- with a **partial** service **able to localize a resource** and so **to request a remote access** to the owner

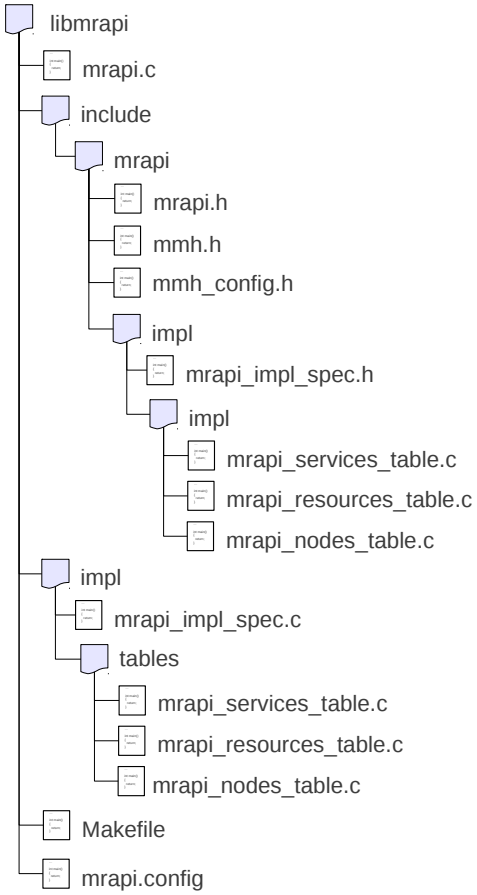


Figure 4.31: MRAPI library file structure

- with a **full** service **able to localize a resource** and so **to request a remote access** to the owner but also **to manage resources locally**

4.5.7.2 Local tables

In order to store all necessary information to communicate between nodes, three local tables are created *(Fig. 4.32)*: a services table which remains the same all along the application execution and so is defined as a static table, a resources table which is updated each time a new global resource is created and a nodes table which is updated each time a node is created inside the domain *(ie. on the core)*. The two latter are managed as dynamic chain lists. Each table is created at the initialization of the core and is shared by every node running on it.

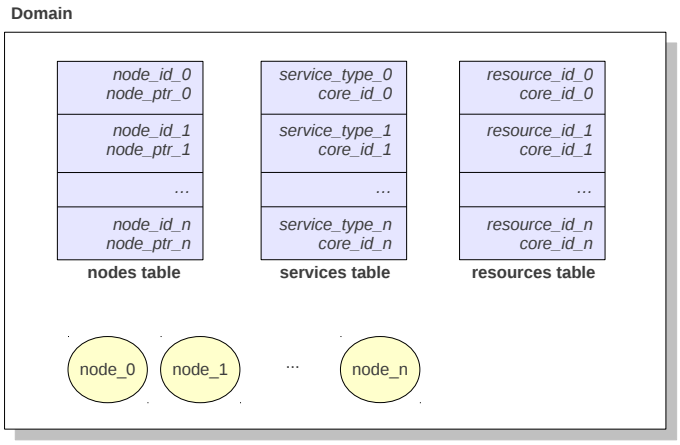


Figure 4.32: MRAPI local tables

4.5.7.3 HAL communication support

To enforce the synchronization between the different cores, we need to rely on a multicore communication API which allows to send messages from one core to another. This API must offer low-level primitives to send and receive these messages. In order to lower the global footprint of the MRAPI implementation, this communication layer is implemented at the HAL level.

4.5.7.4 Remote call management

When a remote call is performed, the message is gathered by a special thread running on each core. Connected to the message acknowledgement mechanisms provided by the implementation, this thread wakes up, create a replica which will process the call and go sleep until the call is completed or another message arrived *(Fig. 4.33)*.

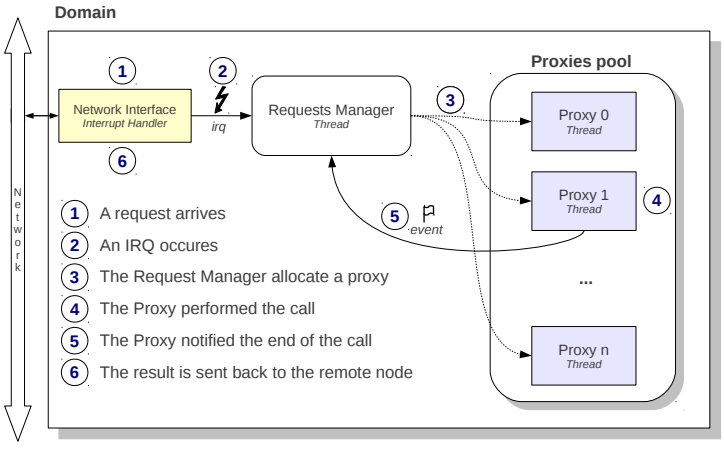


Figure 4.33: Requests management proxies

4.6 Conclusion

In this chapter, we defined the specification of an ideal operating system which would be able to manage a heterogeneous reconfigurable system-on-chip. Regarding the multicore and heterogeneity issues, this operating system have to provide simple communication mechanisms and above all, be enough flexible to efficiently use the dynamicity brought by the Dynamic and Partial Reconfiguration.

To manage it, we relied on the MRAPI specification which provides a simple API on top of the operating system. This API allowed us to implement a flexible server mechanism to adapt the set of services provided by the operating system to the core is running on. Moreover, the modular compilation of the MutekH operating system we chose as a basis is well suited to enforce an easy implementation of a design space exploration tool.

In the next chapter, we evaluate the performance of the proposed solution when integrated in a full heterogeneous and dynamically reconfigurable System-on-Chip.

Chapter 5

Application deployment

Contents

In this chapter we implement a set of features exposed in the previous chapters which characterize an HRSoC, and we build a demonstration platform in an incremental way in order to detail these different features. We give experiment results about the components of the system, including data transfer and system calls timings as well as memory footprints for the software architecture. We also provide timings and resource usage for the hardware one.

5.1 Introduction

A demonstration platform *(Fig. 5.1)* is built to highlight the different communication and abstraction mechanisms provided by this operating system dedicated to reconfigurable platforms which allows to take advantage of the dynamic and partial reconfiguration technology.

This platform is composed of a couple of Microblaze processors, each one being master on its own PLB bus. Also, both have access to a DDR2 memory, an interrupt controller is used to notify the reception of a message by the PLB-NoC bridge and a timer gives the operating system tick and the ability to process timing

measurements.

Specifically, the second Microblaze has an access to the FaRM ICAP controller which allows it to partially reconfigure the FPGA. It has also an access to a DVI controller and is responsible for displaying the processed video. In the frame of this demonstration, the video will be stored into the DDR2 memory.

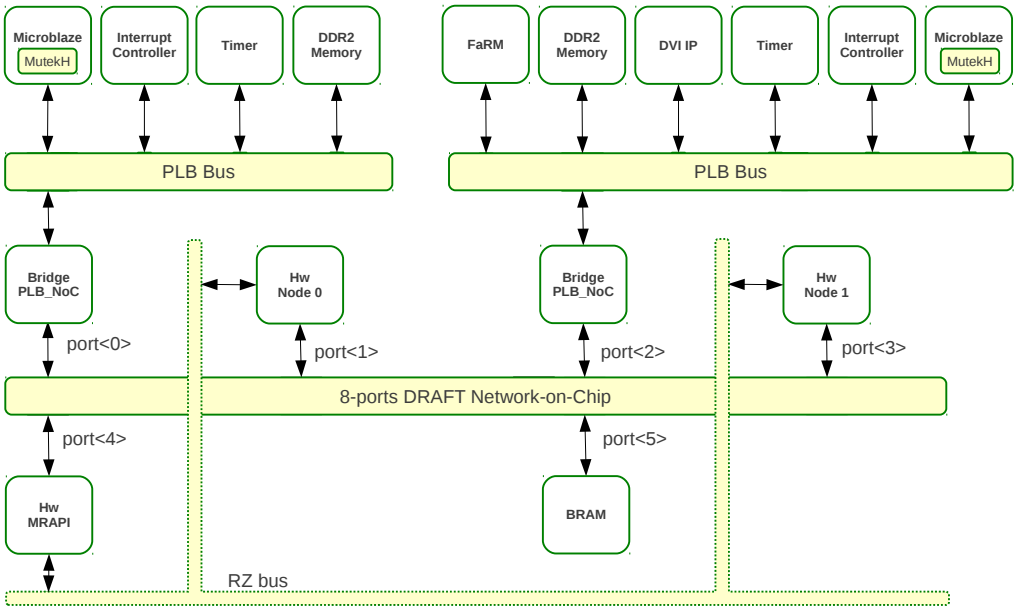


Figure 5.1: Demonstration platform

Two bridges have been instantiated to permit the Microblaze processors to communicate through the 8-port Draft NoC. On the bottom of this NoC, in addition to these bridges, two hardware nodes can be hosted, namely *Hw Node 0* and *Hw Node 1*. On the top of the NoC, the Hw MRAPI module is connected to these hardware nodes by a dedicated bus *(RZ bus on Fig. 5.1)* and a Block RAM connected to the port 5 allows to exchange small amount of data *(8KB)*. All experiments have been realized on a Virtex 5 LX110 Development Board *(xc5vlx110)* designed by Avnet.

5.2 Platform building

5.2.1 Microblaze platform

We start the building of our platform with a simple Microblaze system *(Fig.5.2)*. The instruction and data codes of the application are stored in the Block RAM of the Microblaze. We realized a port of the MutekH operating system on this processor. On top of this operating system, we add the MRAPI layer described in the

Chapter 4 and ported on the MutekH operating system. The memory footprint of each one of these layers is specified in the Table 5.1.

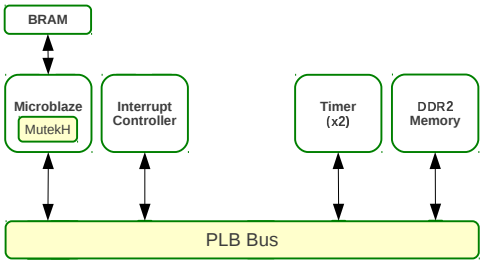


Figure 5.2: Microblaze platform

Component	*Memory footprint*	*Overhead*
MutekH	56392 Bytes	0
with MRAPI	57592 Bytes	1200 (2.12%)
with application (1 node)	58080 Bytes	1688 (2.99%)
with application (2 nodes)	58104 Bytes	1712 (3.03%)
with application (3 nodes)	58112 Bytes	1720 (3.05%)
with application (8 nodes)	58128 Bytes	1736 (3.07%)

Table 5.1: Software layers footprints

The bigger amount of memory that we can provide in the local BRAM of the Microblaze processor is 64 KB. Even if the memory footprint is lower than this capacity, stack and heap overflow can occur when creating new threads. This is why in the next steps in which the MRAPI layer is included, we consider the application code to be stored in the DDR2 memory because its storage capacity is considerably more important. Otherwise, the application is considered to be stored in the local BRAM.

Table 5.2 gives an overview of the latencies generated by the storage of the program and data codes into an internal BRAM memory, an external SRAM memory or an external SDRAM memory for the same application:

5.2.2 Read and Write timings

Memory accesses:

To abstract the heterogeneity of the application, specific communication mechanisms have to be implemented, both in hardware and software, especially the access to the system memory distributed all over the platform. Memories include external

Storage Memory	*without Cache*	*with Cache*
BRAM	4.080 ms	960.168 us
SRAM	57.604 ms	17.282 ms
SDRAM	93.703 ms	29.283 ms

Table 5.2: Code execution time for a Microblaze processor (ML506 @ 125 MHz)

DDR2 memory and local Block Rams connected on the top of the NoC *(Fig. 5.3)*.

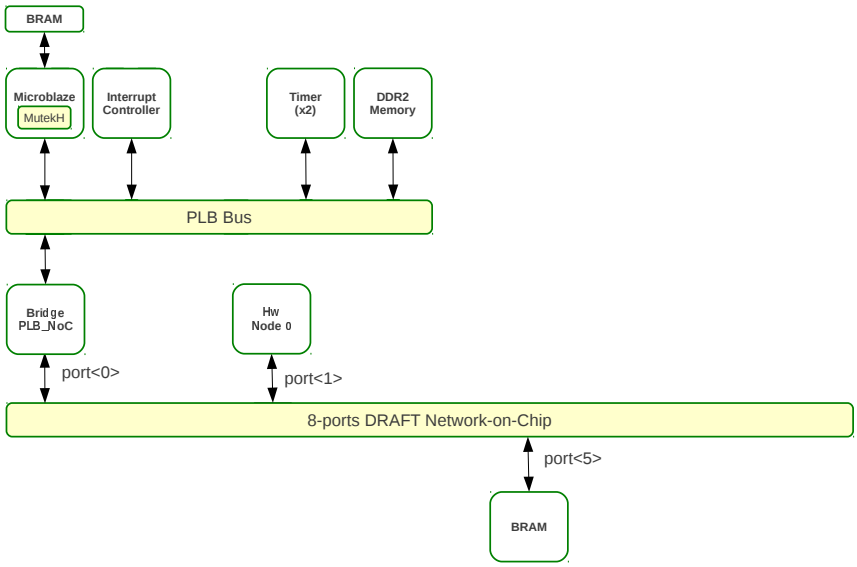


Figure 5.3: Read and write test platform

The architecture of the bridge developed to interconnect the PLB-based Microblaze system with Draft, the network-on-chip implemented by CAIRN, is depicted in Figure 5.4. This one is based on the PLB Master Burst IP [Xilinx 2010b] and allows half-duplex transfers between the PLB to the NoC interfaces and data copies between two buffers mapped on the PLB memory. These transfers are controlled by the Master Command FSM which is driven by the processor using control registers.

To this platform, we add a hardware thread instance *(Hw Node 0)* which can also perform direct read and write access to the BRAM NoC memory and indirect ones to the DDR2 memory passing through the bridge. Indirect because the bridge being controlled by the processor, data packet can continue to the DDR2 memory only if this one enables it.
Tables 5.3 and 5.4 detail the data transfer timings between software or hardware nodes and the platform memories. The program code is stored in the local BRAM

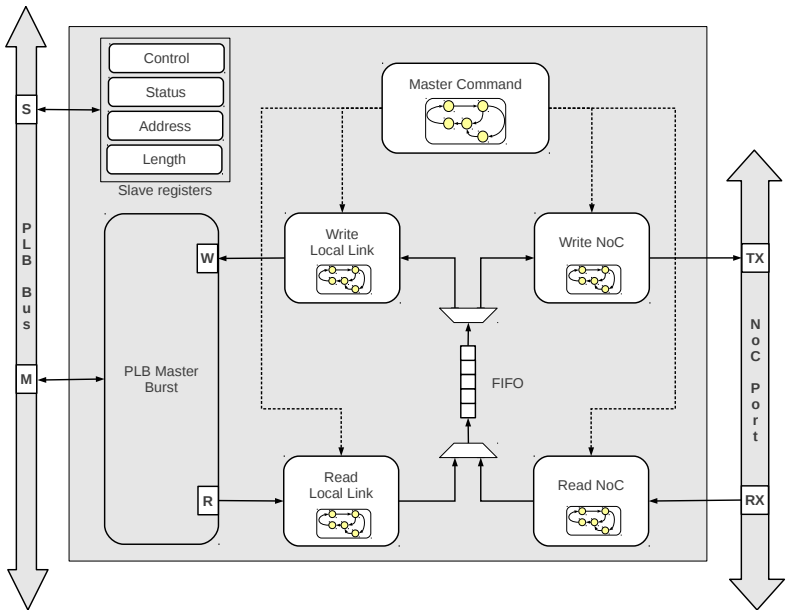


Figure 5.4: Bridge PLB-NoC architecture

of the Microblaze.

Node	***DDR2*** *(cycles)*	***Throughput***	***Bram-NoC*** *(cycles)*	***Throughput***
Sw : write 1 KB	4390	17.11 MB/s	1961	38.30 MB/s
Sw : write 2 KB	8742	17.18 MB/s	3637	41.30 MB/s
Sw : write 4 KB	17446	17.22 MB/s	6988	42.99 MB/s
Sw : write 8 KB	34854	17.24 MB/s	13721	43.79 MB/s
Hw : write 1 Ko	2152	34.9 MB/s	1354	55.48 MB/s
Hw : write 2 Ko	3691	40.7 MB/s	2667	56.33 MB/s
Hw : write 4 Ko	6735	44.6 MB/s	5290	56.80 MB/s
Hw : write 8 Ko	13443	44.7 MB/s	10494	57.26 MB/s

Table 5.3: Timings in cycles to write into platform memories

Table 5.5 details the timings for the read and write transactions from a hardware thread to a BRAM connected on the top of the NoC. Timing measurements follow the data path visible in the description of the Network Interface of the hardware thread *(cf. Section 2.3.3.5)*.

This table shows that the time needed to process a read transaction is more important than to process a write transaction. This is due to the fact that there is a delay needed when setting the address before to get the data from the BRAM

Node	*DDR2* *(cycles)*	*Throughput*	*Bram-NoC* *(cycles)*	*Throughput*
Sw : read 1 Ko	7658	9.80 MB/s	3016	24.90 MB/s
Sw : read 2 Ko	15278	9.83 MB/s	5446	27.58 MB/s
Sw : read 4 Ko	30518	9.84 MB/s	10306	29.15 MB/s
Sw : read 8 Ko	61002	9.85 MB/s	20504	29.30 MB/s
Hw : read 1 Ko	1956	38.40 MB/s	2154	34.87 MB/s
Hw : read 2 Ko	3636	41.32 MB/s	4254	35.31 MB/s
Hw : read 4 Ko	6924	43.39 MB/s	8447	35.57 MB/s
Hw : read 8 Ko	13685	43.91 MB/s	16845	35.67 MB/s

Table 5.4: Timings to read from platform memories

but also an additional 1-cycle delay is introduced to ensure that the NoC is ready to transfer a data. This latency has been added to avoid timing failures and data loss but can be optimized in some cases.

Operation	*Time*
Push wr. req. by User FSM	6 *cycles*
Pop wr. req. by Packetizer	7 *cycles*
Process wr. req. by Packetizer and DMA	8 *cycles*
Push rd. req. by User FSM	8 *cycles*
Pop rd. req. by Packetizer	9 *cycles*
Process rd. req. by Packetizer and DMA	10 *cycles*
Write 32-bits word (Full process)	16 *cycles*
Read 32-bits word (Full process)	21 *cycles*

Table 5.5: Network Interface Communication Measurements

Node Communications:

In addition to memory accesses, the platform offers to the nodes the ability to initiate direct communications. *Table 5.6* presents the network-on-chip interface performances. The protocol used to abstract the heterogeneity of the communications has been introduced in the Section 2.4.2.

We can see that the communication between software and hardware nodes is limited by the bridge performances but globally, the DMA mechanism implemented inside the bridge provides efficient and fast communications between the different domains.

Sender	*Receiver*	*Timing*	*Throughput*
Sw domain	Sw domain	2301 *cycles*	32.64 MB/s
Sw node	Hw node	2069 *cycles*	36.30 MB/s
Hw node	Sw node	2138 *cycles*	35.13 MB/s
Hw node	Hw node	1341 *cycles*	56.01 MB/s

Table 5.6: NoC Send timings for 1 KB data

5.2.3 System calls

The platform used to test the different system calls configuration is illustrated in *Figure 5.5*. The system is composed of three domains: two software ones represented by the domains 0 and 1, and a hardware one as the number 2.

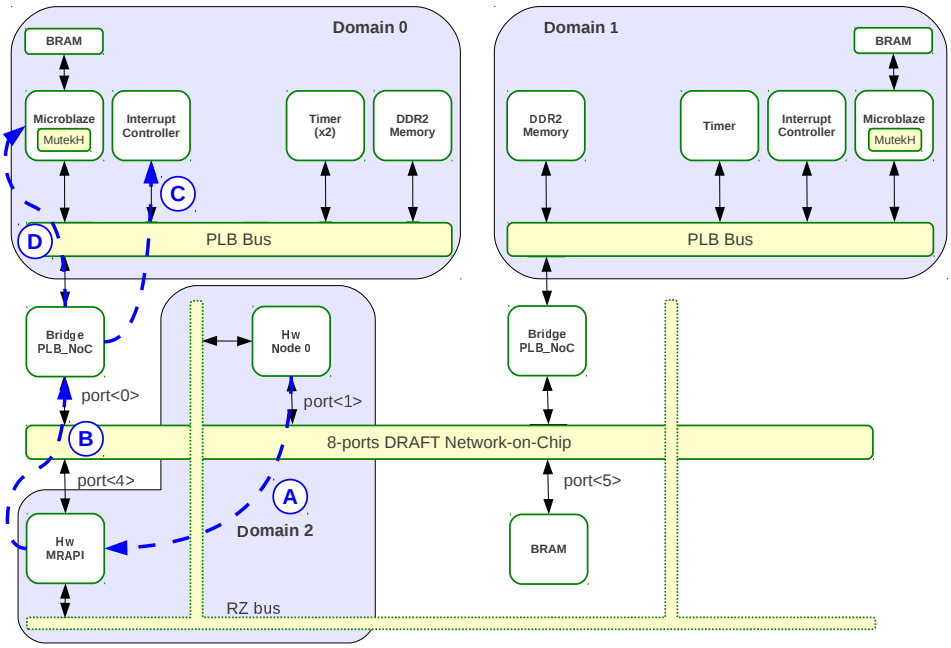


Figure 5.5: Hardware platform used to test system calls procedures

Once communication mechanisms is set up, we can add the upper layer of the communication infrastructure. In this way, we extend the platform including the Hw MRAPI module *(Fig. 5.6)* on top of the NoC. In phase A, the hardware node processes the system call. This call is then encoded and transmitted through the Network-on-Chip by the Hw MRAPI module *(Phase B)*. In phase C, the message is received by the software node on the bridge inputs and an interrupt is launched to the Microblaze processor. Finally, the processor gets the message from the bridge and handles the request *(Phase D)*.

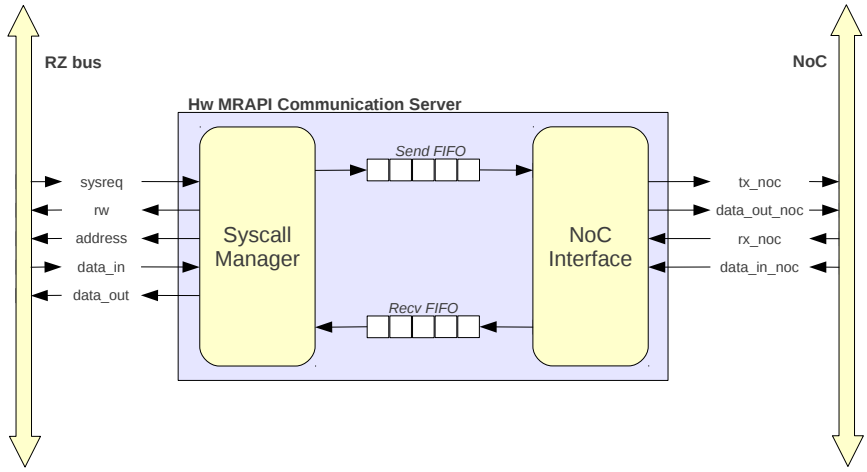


Figure 5.6: Hardware MRAPI global architecture

The Hardware MRAPI module is used to abstract the heterogeneous communication, especially the access to the synchronization mechanisms provided by the operating systems distributed all over the platform *(Fig. 5.5)*. It is connected as a master on the RZ bus and is responsible for the boot and the initialization of the hardware nodes. It can be likened to a MRAPI communication server. The Network Interface gives it a way to send or receive MRAPI requests. The Syscall Manager formats and decodes these requests. The resource usage of this component is described in *Table 5.7*.

Component	***Reg.***	***LUTs***	***BRAMs / FIFOs***	***DSP***	***Freq. (MHz)***
Hw MRAPI	312	541	2	0	251.256

Table 5.7: Hw MRAPI Resources usage

At the software side, times needed to initialize a node, to get its generated node ID, and to initialize the mutex attributes are given in Table 5.8. The call to the different primitives of MRAPI often involves calling the *mrapi_node_id_get* primitive. The two others are only called when initializing the system.

System calls can be of two types: either local or distant. In the case of a local call, we measure the time taken to process system call using the genuine primitives provided by the operating system and the overhead brought by the MRAPI layer *(Table 5.9)*:

The main overhead is due to the management of the node tables used to allocate

Primitives	*Sw Node (DDR2)*
mrapi_initialize	1354 *cycles* (17.6 us)
mrapi_node_id_get	1368 *cycles* (17.7 us)
mrapi_mutex_init_attributes	1054 *cycles* (13.7 us)

Table 5.8: Timings to locally initialize a node

Primitives	*MutekH*	*MRAPI*	*Overhead*
mutex_create	345 *cycles* (4.4 us)	3383 *cycles* (43.9 us)	x9.8
mutex_lock	846 *cycles* (10.9 us)	3018 *cycles* (39.2 us)	x3.5
mutex_unlock	1756 *cycles* (22.8 us)	2888 *cycles* (37.5 us)	x1.6
mutex_delete	184 *cycles* (2.3 us)	3580 *cycles* (46.5 us)	x19.4

Table 5.9: Timings to access a local Mutex resource

or deallocate a new mutex resource. When creating a mutex, we have to find a free place in the table and to initialize the resource structure. After this process, the node ID is sent to the other domains of the platform. Moreover, the specifications imposes several parameter checks and error management for each system call.

In the case of the distant call, we add the time taken to send a request message to the owner domain of the resource, and to receive the return values. Also, different couples of Node sender / Domain receiver are possible and implemented: a software node sends a request to a software domain, or a hardware node sends a request to a software domain. System calls timings are depicted in *Table 5.10*.

Primitives	*Sw Node to Sw Domain*	*Hw Node to Sw Domain*
mutex_create	752 116 *cycles* (9.77 ms)	740 611 *cycles* (9.62 ms)
mutex_lock	652 898 *cycles* (8.48 ms)	576 234 *cycles* (7.49 ms)
mutex_unlock	767 580 *cycles* (9.97 ms)	658 526 *cycles* (8.56 ms)
mutex_delete	767 109 *cycles* (9.97 ms)	729 167 *cycles* (9.47 ms)

Table 5.10: Timings to access a remote Mutex resource

Details about these timings are given in *Table 5.11* for the second case of the *Table 5.10*, where a hardware node requests a resource located on a software domain. The different sections when crossing over the MRAPI layers are illustrated in *Figure 5.7*.

The main overhead is due to the preemption latency between each threads processing the system call, that is to say the Request Manager thread and the allocated Proxy thread. This latency depends on the operating system tick. In our case, we

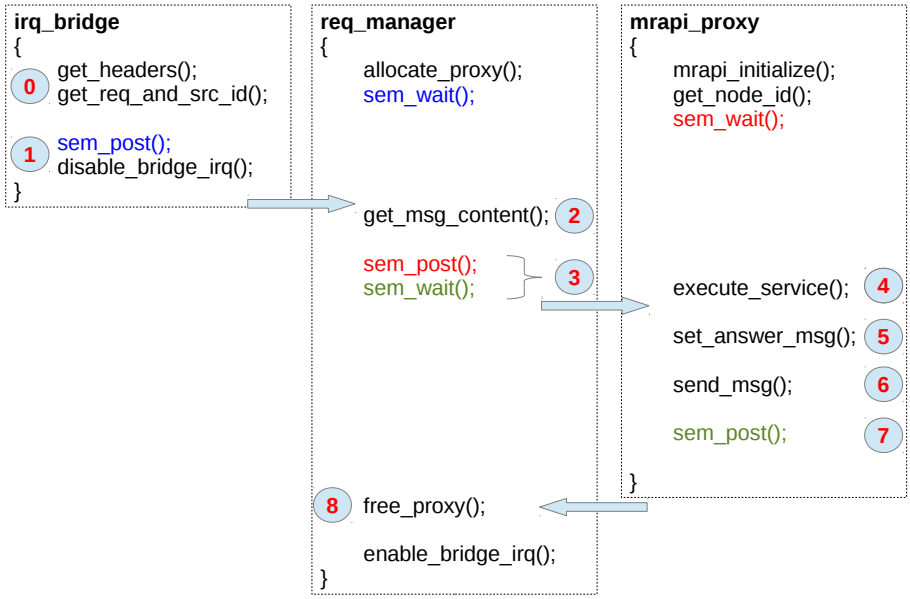


Figure 5.7: MRAPI remote call sections

cannot set a tick lower than 3 ms.

A solution to overcome this issue would be to target a more recent technology such as the Zynq platforms. With hardware dual-core processors, it would be possible to get a higher running frequency and have each thread running on a different core.

The two following cases have not been implemented yet: a software node sends a request to a hardware domain and a hardware node sends a request to a hardware domain, because no hardware service has been implemented in this platform in the frame of this thesis.

5.2.4 Hardware Threads encapsulation

The resources measurements for the static part of the hardware node when implementing the pipeline mechanisms are illustrated in the *Table 5.12*.

On top of these abstraction layers, an application composed of software and hardware threads can be deployed. In the next section, application scenario and partitioning choices are first described and, to conclude this chapter, application performances and results are given.

Sections	*Create*	*Lock*	*Unlock*	*Delete*
Hw MRAPI request	28 *cycles* (0.364 ms)	29 *cycles* (0.377 ms)	28 *cycles* (0.364 ms)	24 *cycles* (0.312 ms)
0	3747 *cycles* (48.7 μs)	3745 *cycles* (48.6 μs)	3772 *cycles* (49.0 μs)	3733 *cycles* (48.5 μs)
1	3537 *cycles* (45.9 μs)	3533 *cycles* (45.9 μs)	3538 *cycles* (45.9 μs)	3532 *cycles* (45.9 μs)
2	14011 *cycles* (182.1 μs)	16396 *cycles* (213.1 μs)	14143 *cycles* (183.8 μs)	11607 *cycles* (150.8 μs)
3	2606 *cycles* (33.8 μs)	2594 *cycles* (33.7 μs)	2599 *cycles* (33.7 μs)	2595 *cycles* (33.7 μs)
4	3775 *cycles* (49.0 μs)	3431 *cycles* (44.6 μs)	3252 *cycles* (42.2 μs)	3985 *cycles* (51.8 μs)
5	1627 *cycles* (21.1 μs)	1631 *cycles* (21.2 μs)	1642 *cycles* (21.3 μs)	1627 *cycles* (21.1 μs)
6	19590 *cycles* (254.6 μs)	17412 *cycles* (226.3 μs)	17541 *cycles* (228.0 μs)	17408 *cycles* (226.3 μs)
7	2602 *cycles* (33.8 μs)	2609 *cycles* (33.9 μs)	2599 *cycles* (33.7 μs)	2590 *cycles* (33.6 μs)
8	489 *cycles* (6.3 μs)	494 *cycles* (6.4 μs)	495 *cycles* (6.4 μs)	480 *cycles* (6.2 μs)
Total Sw	51984 *cycles* (675.7 μs)	51845 *cycles* (673.9 μs)	49581 *cycles* (644.5 μs)	47557 *cycles* (618.2 μs)
Hw Mrapi get returns	1106 *cycles* (14.3 μs)	963 *cycles* (12.5 μs)	963 *cycles* (12.5 μs)	963 *cycles* (12.5 μs)
Process returns value	10 *cycles* (0.13 μs)	8 *cycles* (0.10 μs)	8 *cycles* (0.10 μs)	8 *cycles* (0.10 μs)

Table 5.11: Detailed timings to access a remote Mutex resource

5.3 Tracking application

5.3.1 Presentation

The application deployed on the demonstration platform is a target tracking application whose the genuine version is illustrated in *Fig. 5.8*. This application is responsible for detecting and tracking targets in an infra-red video stream. In the frame of this demonstration, the spatial resolution has been set up to 128x128 pixels per frame.

The application is divided into four static nodes, and a dynamic one. The first thread of the static part corresponds to the acquisition of the data from the camera *(Acquisition)*. It is followed by the target detection thread *(Detection)*. The third

Component	*Reg.*	*LUTs*	*BRAMs*	*DSP*	*Freq. (MHz)*
OS Interface	38	73	1	0	956.938
System FSM	4	6	0	0	781.250
Sync. ctrl	104	156	1	0	284.333
FU Recv	68	226	0	0	248.369
FU Send	105	175	0	0	299.850
Token Counter	6	12	0	0	448.430
Token Checker	9	15	0	0	534.474
FIFO Req. ctrl	37	60	1	0	284.333
FIFO Sync. ctrl	27	34	1	0	381.679
Sync. module	311	467	3	0	284.333
Hw Task static	562	939	6	0	271.370

Table 5.12: Hardware Thread Resources Usage.

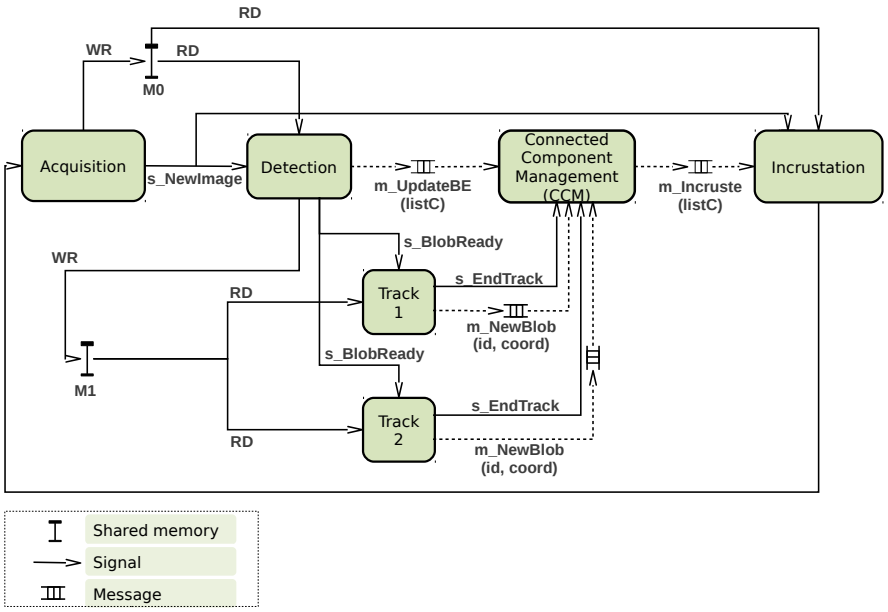


Figure 5.8: Target Tracking Application

thread gets the results from the tracking threads and ensures the coherency within a list of the current tracked targets *(CCM)*. A last thread asks for this list and displays the bounding box of each target into the input image *(Incrustation)*.

In the version that we implemented, the dynamic part of the application is represented by the tracking threads, each one responsible for maintaining the coordinates of one of the detected targets in the video *(Tracking)* by computing the Continuously Adaptive Mean shift *(Camshift)* algorithm [Cheng 1995]. As a result, they

provide the CCM thread with the bounding box coordinates of the target they are tracking. In order to emphasize the management of the dynamicity provided by the platform, we chose to implement the different tracking threads in hardware.

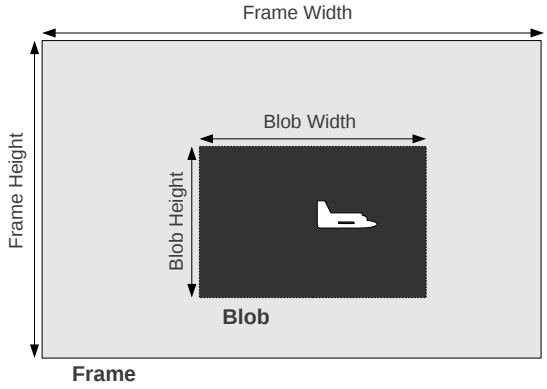


Figure 5.9: Binary Long Object (Blob)

Specifically, the core responsible for the computing is implemented in a hardware thread, and a software one is in charge of both the initialization of this core and of the data transfer. Data transferred between the two threads consists in a *Blob* or *Binary Long Object*, which is a sub-frame supposed to contain the target to be tracked. Therefore, for each frame, there will be one blob per target *(Fig. 5.9)*.

5.3.2 The Camshift IP

As said previously, a blob is processed by a hardware task which implements the Camshift algorithm. The Camshift IP is implemented as pictured in the Figure 5.10. This IP receives from the NoC the blob containing the target assigned by the Detection thread. This reception is realized by the dedicated Functional Unit *(FU Recv)*, which stores received data inside the thread memory *(M1a or M1b)*.

Then the *FU Recv* sends a synchronization token to the Camshift core embedded in the Camshift Functional Unit *(FU Camshift)*. The core computes the target coordinates before to save it with the notification of convergence *(flag indicating if the target has been locked)* inside the second buffer *(M2a or M2b)*. After computing, the FU Camshift sends a synchronization token to the FU responsible for sending the results.

Finally, the last Functional Unit *(FU Send)* reads the values stored in the second buffer and sends it to the software domains. After reception, Domain 1's processor uses these coordinates to insert the bounding box, whereas Domain 0's processor acquires a new blob inside the current frame, depending on the received results.

All these synchronizations are managed by the Synchronization Module *(Sync. Module in Fig. 5.10)*, and ordered by the User FSM. Some data is directly exchanged

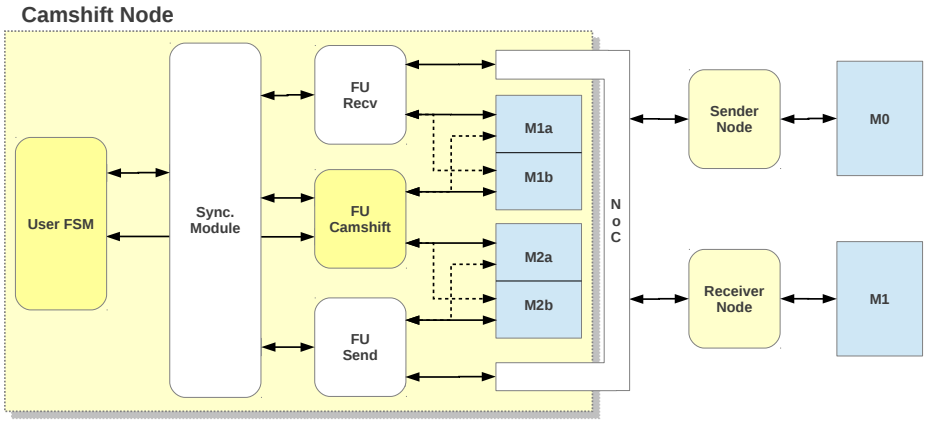


Figure 5.10: Pipelined Camshift hardware node

between this User FSM and the FU Camshift in order to parametrize the reception request as we cannot know the size of the next blob before the end of the current computation. This is why, as depicted in *Figure 5.11*, we have to stall between the end of the send request and the following cycle of receive-compute-send requests.

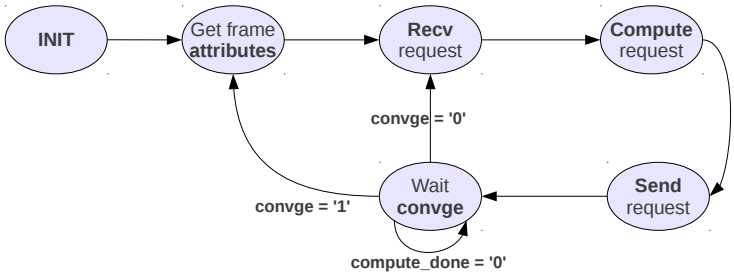


Figure 5.11: Pipelined Camshift User FSM

5.3.3 The DVI IP

The LX110 Development Board that we use for our demonstration platform does not come by default with a video output. In order to display the frames before and after the processing, we complete the platform with a dedicated daughter-board which provides, among other features, a DVI output. This output was driven by a DVI IP that we developed and whose the integration in the demonstration platform is detailed in *Figure 5.12*.

The IP has an access to a VFBC channel *(Video Frame Buffer Controller)* which allows it to directly read data from the DDR2 memory. It also offers a set of five slave registers to the Microblaze of the Domain 1 in order to configure the data

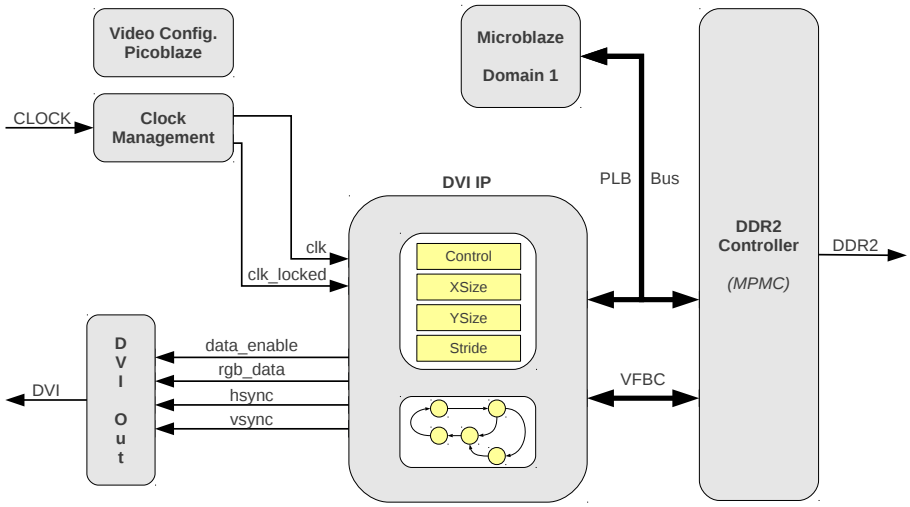


Figure 5.12: Integration of the DVI IP in the Demonstration Platform

transfer between the IP and the DDR2 memory. The VFBC port permits to access 2-Dimensional frames inside the memory and so to dynamically adapt to different resolutions.

5.3.4 Application deployment

The application is deployed on the platform as depicted in *Figure 5.13*. The source video is a gray-scale video and is acquired from the DDR2 memory so we skip the pre-processing of the incoming frame except the binarization of the frame *(Detection process)*. In this video, we consider that we already know the number of targets so we can simplify the Detection process and the *CCM* component only need to manage a static list of these targets.

On the other hand, the second part of the *CCM* process running on *Domain 1* is in charge of the reconfiguration of the hardware tasks regarding the application needs. It means the number of targets per frame and the maximum number of slots dedicated to the Camshift processes *(Fig. 5.14)*.

Figure 5.15 details the software nodes and their synchronizations and interactions inside the platform using the Network-on-Chip communication medium and the operating system services.

On Domain 0, the *Acquisition* node is not created because acquisition does not occur as video frames are pre-loaded in the DDR2 memory. Hence, the first created node is the *Detection* node. The *Detection* node gets frames from the DDR2 *(Static Frames)*. A static pre-initialized table permits the node to know how many targets

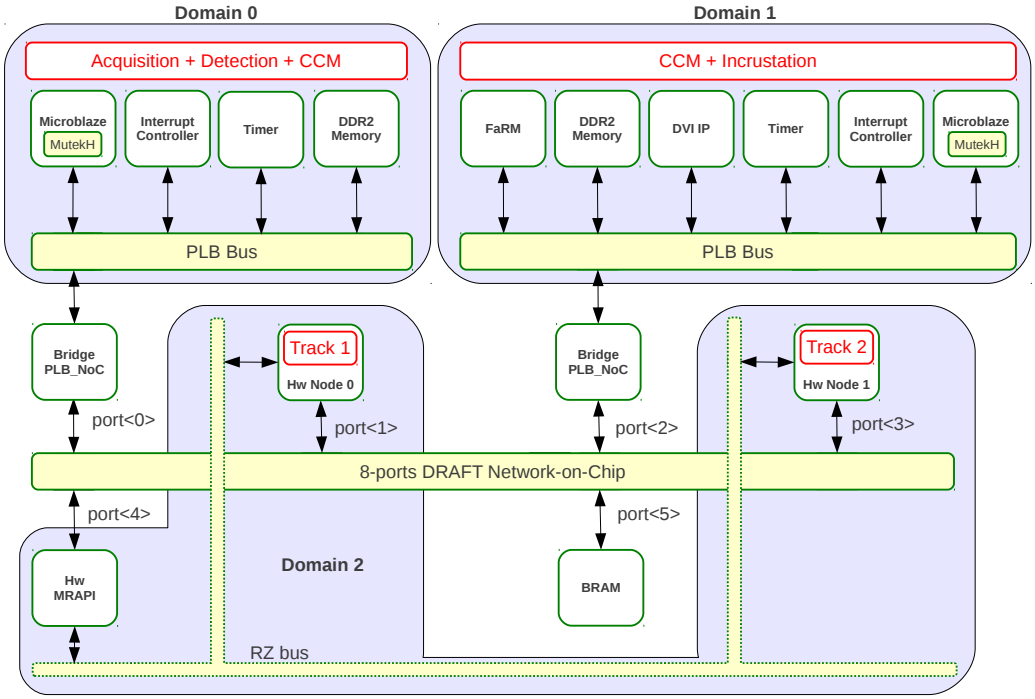


Figure 5.13: Application deployment

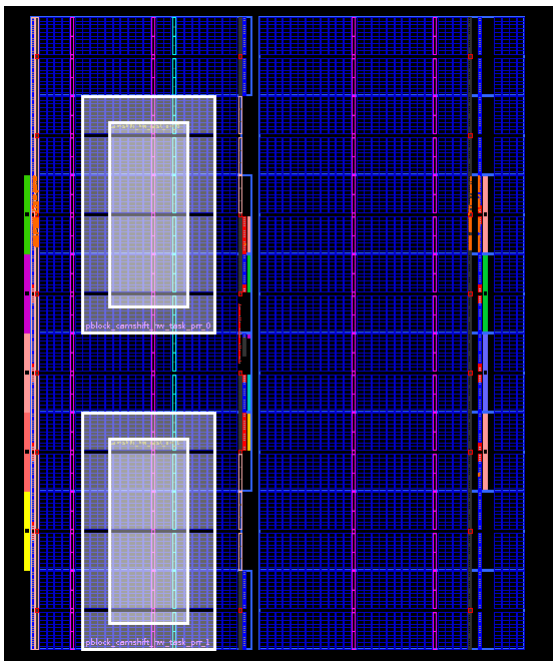


Figure 5.14: Camshift slots (Virtex 5 LX110 device)

are presents in each frame *(Static Targets)*.

Then the Detection node compares the number of detected targets with the number of hardware tasks which are configured at this moment *(Camshift Hw nodes)*. If the number of reconfigured tasks is lower than the number of detected targets and that one or several partition slots are available to host a hardware Camshift task, the *Detection* node unlocks the *Reconfiguration Manager* node to have it performing a reconfiguration request.

Then the *Reconfiguration Manager* node synchronize with the *Reconfiguration Completer* node which is able to drive the FaRM IP to process the requested reconfiguration.

After the reconfiguration request, the *Detection* node allocates a *Tracking* node for each one of the targets that can be processed simultaneously by the different hardware Camshift tasks. Then the *Detection* node unlocks the *Tracking* nodes and waits for a synchronization signal from each one of them to know when the frame processing is completed.

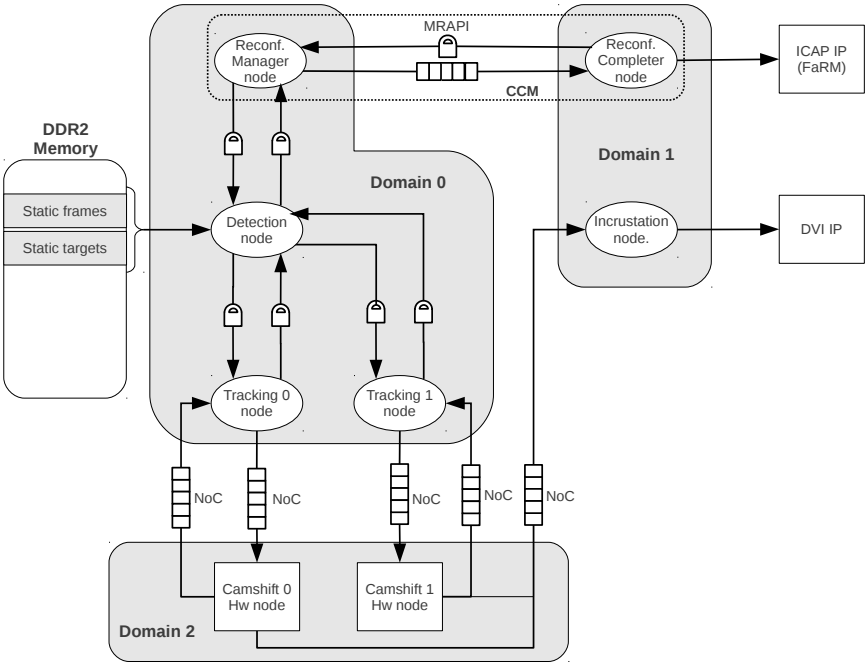


Figure 5.15: Detailed application deployment

The *Tracking* node gets back the information about the target *(initial blob coordinates and hardware node port number)*, then it extracts the blob inside the frame before to send it to the hardware node.

The *hardware node* receives the blob from the Network-on-Chip, processes it and

sends back the coordinates of the new blob to both the Domain 0 and the Domain 1. Results are then collected on the Domain 0 by the software *Tracking* node associated with it, and on the Domain 1 by the *Incrustation* node.

The *Tracking* node compares the new coordinates with the previous ones and checks if the convergence occurred. If this is the case, the node sends a synchronization signal to the *Detection* node to notify the end of the processing. Otherwise, the *Tracking* node loop back and send the new blob corresponding to the previously calculated coordinates.

On Domain 1, the *Incrustation* node modify the buffer used by the DVI IP in order to encompass the detected target, drawing a rectangle around the target. Once every *Tracking* nodes have converged, the *Detection* node starts a new processing round for the next video frame.

5.3.5 Results and performances

Table 5.13 shows the hardware resources needed to host the full demonstration platform on the LX110 device.

Slice Logic Utilization	***Used***	***Available***	***Utilization***
Slice Registers	19980	69120	28%
Slice LUTs	29302	69120	42%
bonded IOBs	113	440	25%
BlockRAM/FIFO	75	128	58%
BUFG/BUFGCTRLs	7	32	21%
DCM ADVs	1	12	8%
DSP48Es	8	64	12%
ICAPs	1	2	50%
PLL ADVs	1	6	16%

Table 5.13: Demonstration Platform resource utilization

Table 5.14 details the hardware resources used by the partially reconfigurable part of the Camshift node. The Hardware Task PRR occupies a partition as large as the slot defined in *Figure 5.14*. The number of resources covered by each slot is indicated in *Table 5.15*. The reconfiguration overhead to load a new Camshift task in this slot, using the FaRM IP running at 75 MHz, equals to 274 228 cycles *(= 3.56 ms (= 2.74 ms at 100 MHz))*.

In our case, this reconfiguration latency can be hidden by the fact that once the *Reconfiguration Manager* node requested the reconfiguration, it is up to the *Reconfiguration Completer* node to process the reconfiguration while the *Detection*

node continue its work with the currently reconfigured hardware nodes. Also, other solution like bitstream file pre-fetching can help reducing this latency.

Component	*Reg.*	*LUTs*	*BRAMs / FIFOs*	*DSP*	*Freq. (MHz)*
User FSM	141	204	0	0	241.354
FU Camshift	193	178	0	1	81.618
Camshift IP	2835	6651	0	0	76.363
Hw Task PRR	**3117**	**6974**	**0**	**1**	**81.618**

Table 5.14: Hardware Thread Resources Usage

Site Type	*Available*	*Required*	*Utilization*
LUT	7680	6973	91%
FD LD	7680	3168	42%
Slice L	1380	1253	91%
Slice M	540	491	91%
DSP48E	24	1	5%
RAMBFIFO36	12	0	0%

Table 5.15: Camshift slot resource utilization

Table 5.16 gives the different timing results of the application. A graphical view of these timings is given in *Figure 5.16*. We can see that the major part of the time is spent in the binarization and the extraction of the blob from the current frame *(get_blob)*.

In a further step, the acquisition chain and the pre-processing of the Detection process are planned to be implemented as hardware IPs. Also, regarding the blob extraction, the use of a processor running at a higher frequency will be sufficient to significantly lower this overhead. With an average time comprised between 74.6 ms and 135.1 ms using pre-binarized frames, the above improvements will permit us to target real-time performances.

5.4 Conclusion

This chapter detailed the design steps of a partially reconfigurable platform. We built a heterogeneous platform system composed of different types of processing units. The computational element includes both general purpose processors and

(Domain) Section	*Cycles*	*Time*
(0) **Binarization**	36923076	480 ms
(0) **Detection**	1068	13.8 μs
(0) **Check slots**	4	52 ns
(0) **Alloc. target**	2409	31.3 μs
(0) **Send attr.**	6195	80.5 μs
(0) **Init blob**	207188	2.69 ms
(0) **Get blob**	11530 - 4791544	149.9 μs - 62.2 ms
(0) **Send blob**	6515 - 8706	84.6 μs - 113 μs
(0) **Recv data**	4805	62.4 μs
(0) **Check cvge**	1446	18.7 μs
(1) **Dvi init**	289	3.7 μs
(1) **Display**	168	2.1 μs
(1) **Wait result**	183 - 4687472	2.3 μs - 60.9 ms
(1) **Incrust**	31304	406 μs
(1) **Display**	376	4.8 μs
Per frame bypassing binarization	–	74.6 ms - 135.1 ms

Table 5.16: Application timings

dedicated accelerators.

The first step was to provide a low-level communication layer, permitting the user to proceed simple read and write transactions. This former layer has been evaluated and a focus has been set on the heterogeneous communication.

As expected, the communication between hardware nodes is more efficient than between software components. Regarding heterogeneous communication, the conception of the DMA capable communication bridge is a good trade-off to take advantage of the heterogeneous processing without being penalized by the communication overhead.

The second step rose up the abstraction layer to the operating system level. In this part, we evaluate the hardware thread encapsulation which allows them to process system call. This ability is provided by the MRAPI layer, a multicore communication API ported on both the software and the hardware domains of the platform. The overhead provided by this additional layer must be evaluated according to the advantage of a totally decentralized operating system service set.

Again, regarding the communication issue between heterogeneous components, this abstraction layer allows to consider a flexible mapping of the application tasks and so to optimize the data transfer between the different nodes.

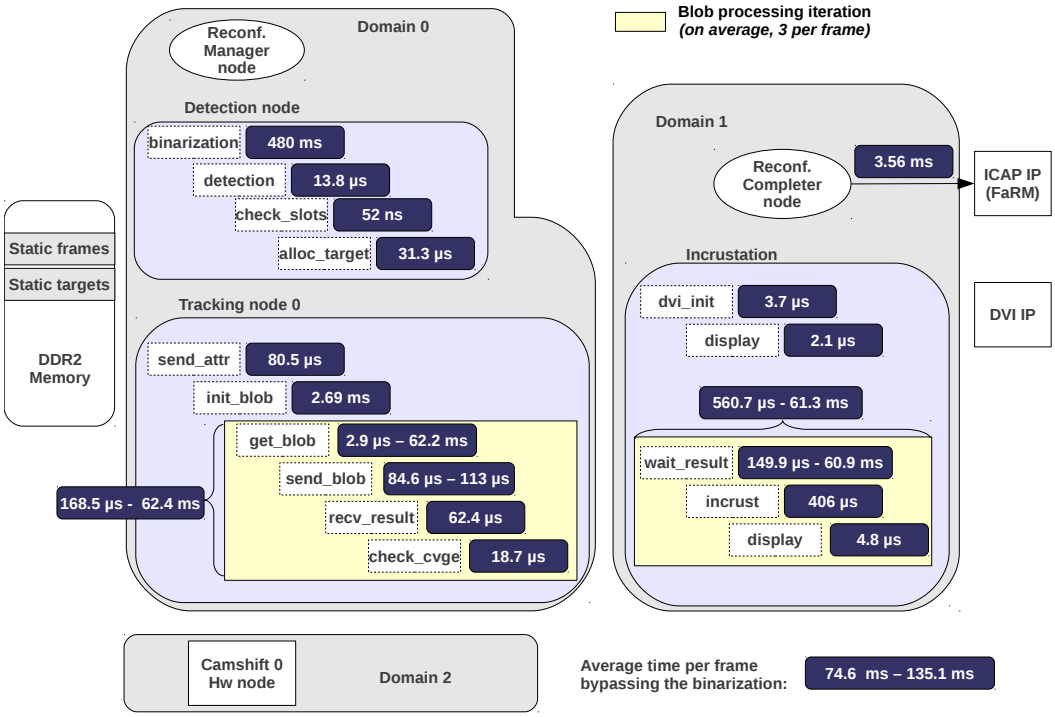


Figure 5.16: Detailed application deployment

Finally, all these communication mechanisms open up the way to the build of a heterogeneous platform with partially reconfigurable ability. The reconfiguration is proceed using the FaRM IP and allows to dynamically load any hardware threads of the application.

The different features of this platform are illustrated by the deployment of a tracking application. This deployment is an important step in the vision that we have of the future of the embedded systems, especially the image processing, the multimedia and the high performance computing systems.

This implementation is effectively a milestone towards the realization of an HRSoC capable to host software tasks and hardware relocatable components, providing a common interface to facilitate both the communication and the mapping. We already planned future work in order to improve the performances of this HRSoC and make it a generic platform allowing to deploy and evaluate real-time applications on a physical system.

Chapter 6

Conclusions

6.1 Summary

6.1.1 Discussion

In this thesis, we discussed the programmability issues encountered when designing reconfigurable systems. From the user point of view, who is considered to be an application developer, either a software one or a hardware one, these issues concerns the management of the heterogeneity and the ability to take advantage of the flexibility offered by the dynamic and partial reconfiguration technology.

To solve these issues, we have turned towards a solution which would provide a programming model fitting on each kind of processing cores embedded in the system. These cores are the processor cores and the reconfigurable IPs.

An important point when designing this kind of platform is to let the user the possibility to choose on which core each one of the functions or tasks of its application can be mapped. This step of mapping should be as flexible as possible in order to allow an efficient design space exploration and so to be adaptable to a larger number of platforms.

Although the main feature of this programming model would concentrate on the task communication, this model should be extended to the services provided by the operating system. This constraint is linked to the fact that the operating system model is widespread and that we need to support legacy model.

Moreover, each task of the application should be able to access any services provided by the system. Obviously, these accesses should be limited by timing considerations and latencies overheads due to the physical location of the operating system services regarding the tasks ones, but again, this choice should be left to the user.

6.1.2 Key contributions

Addressing these issues respecting our constraints required the realization of a new operating system dedicated to the reconfigurable systems, and especially to the dynamically and partially reconfigurable systems.

The first step has been to leverage the hardware description to the same level as the software ones. Another constraint was to keep using classical HDL tools provided by the FPGA manufacturers. In concrete terms, the objective was not to create a new language but to provide a new abstraction level to manipulate the hardware component. At this level, the interaction between a software task and our hardware task model relies on a common interface which provides an access to OS primitives for the hardware tasks and gives us the ability to swap the way we map a task, either in software or in hardware.

This programming model has been coupled with the integration of a preemption service inside the operating system. This service is responsible for managing the save and the restoration of the hardware tasks context. It relies on two features: in one hand the knowledge of the internal structure of the configuration files, and on the other hand the re-use of an existing IP *(FaRM)*, which permits us to improve the reconfiguration process.

In addition, a relocation service has been implemented. In this way, we investigate a new solution based on the reversal of the Isolation Design Flow provided by Xilinx in order to design relocatable hardware tasks. With some additional constraints inserted in the PlanAhead tool and management scripts to automatically insert these constraints, we manage to relocate a hardware tasks from one partition to another one using a unique partial configuration file.

On the software side, an implementation of the MRAPI specification has been done to facilitate the synchronization with the hardware tasks. A set of three different operating servers has been proposed in order to fit with the needs of a reconfigurable platform.

Each one of these servers respectively allows to access directly to a known remote operating resource *(for instance, a Semaphore, a Mutex or a memory buffer)*, to locate and access to an unknown remote resource and to locally create and process a resource. These three types of servers permit to adapt the deployment of the operating system resources regarding the specificity of each one of the cores it is composed of.

Finally, the implementation of a complete heterogeneous and reconfigurable system-on-chip is a good achievement of this programming model on a physical support.

6.1.3 Hypothesis and Limitations

The first hypothesis which is also the first limitation of our operating system is that we consider the number of core to be inferior to sixteen because we decided to specifically target multicore platforms and not manycore systems.

Regarding the communication on the platform, the PLB structure has limit; in

this sense AXI would be better as we could connect the hardware tasks directly to the AXI infrastructure, through the AXI-Stream interface for instance, and so conserve efficient direct communication with hardware and software task without passing through a custom bridge.

Also, when adding new abstraction layers as the MRAPI layer, the price can be lower performances, especially regarding timing overheads which can be important, but the gain obtained in the programmability cannot be neglected.

Finally, the choice of keeping the manufacturer tools in order to perform the relocation imposes us to be dependant of their evolutions, but also to their limits. This is the case when a routing process does not provide the wanted results and that the only solution would be to manually route the conflicting wires. This work can be processed using specific tools like RapidSmith but in our case, knowing these limitations, the solution we propose can be easier and faster under favourable conditions.

6.2 Future Work

In future work, the management of the dynamic and partial reconfiguration in the case study that we used to demonstrate our propositions can be enhanced. Instead of reserving the reconfiguration of the hardware thread only to add or remove a Camshift thread in the platform, we can process functionality switching and implement some of the pre-processing operations in hardware.

Another track that could be interesting in the future is to reconsider the association between an MRAPI node and a hardware thread. Keeping the concept that a software thread is a node running on a processor which is a domain, we can enhance the parallel defining a hardware thread as a domain, which leads to define a node as a partially reconfigurable region *(PRR)*. All PRRs will share the same hardware thread processing unit which allows a PRR to send and receive data, and also to call local or remote services *(Fig. 6.1)*.

Also, a flexible adaptation of the Processing Functional Unit *(FU)* could be realized to allow the user to choose the interfaces required by the functional unit. The choice can be made, for instance, between a memory port, in read or write access, or even both, and a FIFO port. Several combination can be thought in order to make a trade-off between the enhancement of the stream processing with FIFOs, and the storage capacity and random access provided by memory blocks. Then this is up to the user to develop the logic glue which will control the access to the data.

Regarding the relocation issue, we will investigate the Isolation Design Flow which is now available for the 7 series. Larger devices allow more flexibility for the routing engine and so we could increase the success rate of the proposed re-

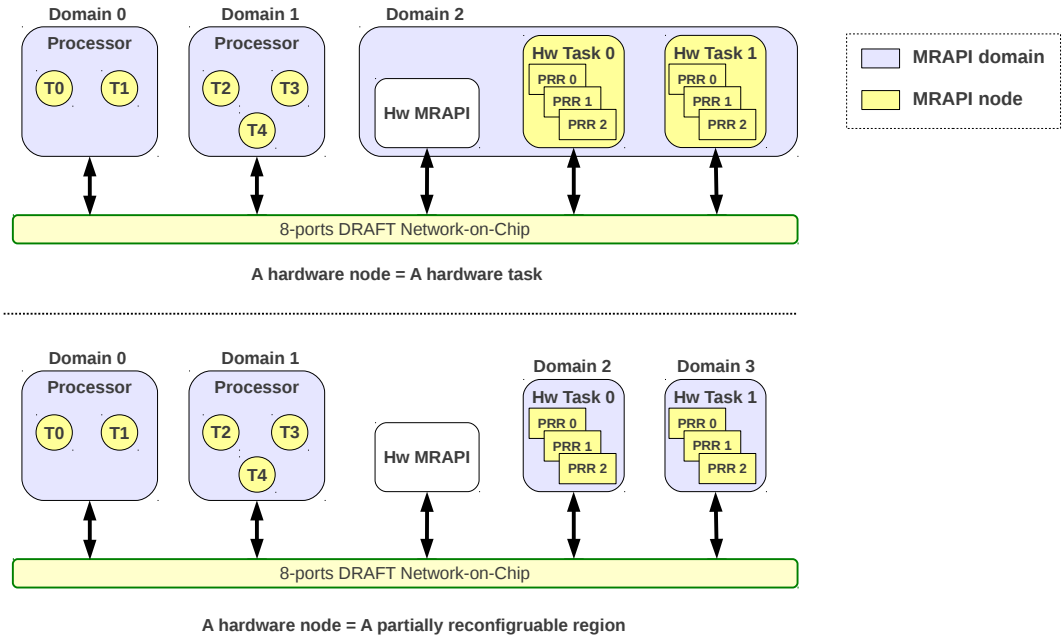


Figure 6.1: Hardware node implementation choices

location flow. This type of design flow leads the way to the conception of very flexible systems-on-chip in which the different tasks can be moved over the platform regarding external parameters. These parameters can be influenced by the power consumption, the heating issues or the communication issues. For the last example, the increasing size of the FPGA can lead to important routing latencies which can be overcome with the displacement of the communicating tasks.

Furthermore, the multiplicity of the proposed features in a single device open the way to the management of complex applications in which the numerous modes of execution and the associated quality of service that must be provided by the system should be handled.

To conclude, beside the technical aspect, a vision of the future of the FPGA, and more specifically of the HRSoC, can be introduced. Today, FPGA manufacturers adopt two different approaches to speed up and facilitate the development process on their devices. On one hand, the High-Level Synthesis *(HLS)* approach, proposed by Xilinx, allows the user to describe a hardware IP in a high-level language and a special synthesizer is responsible for the translation to the RTL level. The advantage of this method is its modularity and the fine-tuning possible using precise synthesis options. The goal is limited to the enhancement of the IP development process.

On the other hand, the OpenCL-based approach offered by Altera permits to describe the whole application in a high-level language and interfaces are automatically created to make the communication between the different parts of the application, either software or hardware, transparent for the developer. In this way, this solution

is more turned toward the high-performance computing but the tuning capacity is limited.

We think that these two approaches are intended to converge and to form a complete design flow allowing to create HRSoC platforms and applications from a high-level model to a true implementation level.

To achieve this, in this thesis we defined a low-level encapsulation of the hardware component in order to support a data-flow programming model, coupled with the classical threading model. The interfaces that have been defined provide a sufficient abstraction level to consider a heterogeneous application to be homogeneous. From here, only few efforts are necessary in order to integrate these components into a complete design flow able to automatically map each part of the application and to ensure their correct communication.

However, even if we want to keep this abstraction to be able to model the whole application, we also want to keep a full control over the hardware implementation. In this way, the design of the hardware thread has been thought to be modular and so it will be possible to automatically generate hardware tasks source codes and interfaces using a dedicated high-level synthesis tools, and so to form a complete design flow.

Appendix A

Network Interface API

Contents

A.1 Supported requests

The two first basics request supported by the Network Interface are the Send and Receive primitives. A Send request consists in sending one or several packets over the network. Packet size is fixed by communication medium design *(ie. the NoC)*. A packet is composed of a header followed by data to transmit. Data and header are represented by 32-bit width flits. A receive request consists in waiting for a packet to come from the network. It is a passive request which involves no transmission from the requesting thread.

The two others supported requests are the Write and Read primitives. A Write request is similar to a Send request except that additional header flits are sent after the two main flits. The main flits are essential to ensure a correct routing inside the network. The first flit contains the sender and receiver port address whereas the second one contains the number of flits included in the packet.

A.1.1 Write request

Packet sending, from a thread to external memory connected on the network.

The procedure *SEND_PROC(data_size, data_ptr, port_addr, buffer_addr)* is both used to send Write and Read commands to a memory connected on the NoC. For a **Write** command, the parameters are the following :

◇ data_size: **size** of the data to **send** in the **internal memory**

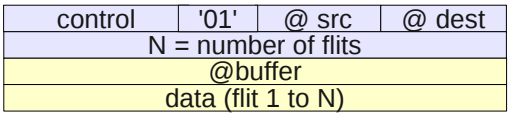


Figure A.1: Write request packet

◇ data_ptr: **pointer** on the data to **send** in the **internal memory**

◇ port_addr: **port identifier** address of the **external memory** connected on the NoC

◇ buffer_addr: **pointer** on the buffer in the **external memory**

A.1.2 Read request

Packet sending, from a thread to an external memory connected on the network. The packet contains useful information for the memory to to read and send back read data. to the thread. In the case of an exchange between hardware threads, because a thread owns its own internal DMA, the flit "@write_buffer" is not used.

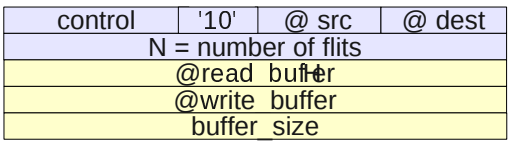


Figure A.2: Read request packet

For a **Read** command, the parameters are the following :

◇ data_size : **size** of the data to **read** in the **external memory** *(buffer_size)*

◇ data_ptr : **pointer** on the data to **read** in the **external memory** *(read_buffer)*

◇ port_addr : **port identifier** address of the **external memory** connected to the NoC

◇ buffer_addr : **pointer** on the buffer used to **write** the read data. Used by the **external memory** to make the response *(write_buffer)*

It gives back the hand to the user immediately after request parameters have been stacked into the FIFO.

A.1.3 Read request response

Reception by a thread, of one or several packet from an external memory; Packets coming after a read request from the thread to this external memory.

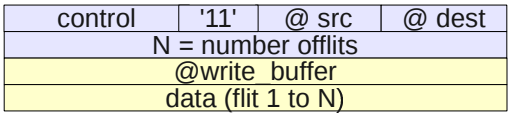


Figure A.3: Read request response

A.1.4 Receive request

RECEIVE_PROC(data_size, data_ptr, port_addr, buffer_addr)

- ◇ data_size : **size** of the data to **receive** from the NoC
- ◇ data_ptr : **pointer** on the buffer used to **write** the received data in the **internal memory**
- ◇ port_addr : **port identifier** address of the sender
- ◇ buffer_addr : **pointer** on the buffer used to **write** the data. Not used by a hardware task, only by the NoC-AHB bridge

It gives back the hand once the depacketizer received the whole data. At this moment, the on_duty_depack signal is cleared.

Appendix B

Hardware CRC

Contents

B.1 Relocation process

As discussed in Section 3.4.3, in order to relocate a bitstream from one partition to another it is necessary to process to the readback of the first partition.

After the readback, the relocation to the other partition is done modifying the FAR value *(Frame Address Register)* contained in the readback bitstream. As during the module execution, some configuration data like the Flips-Flops or the memory contents may change, we need to recompute the CRC for these new values to avoid an ICAP rejection.

Notice: Disabling the CRC control is possible using the default value 0x0000DEFC. However, for safety reason, this is highly inadvisable.

B.2 CRC computation

According to [Xilinx 2006], the CRC computation on the Virtex 4 devices is not processed using all data written to configuration port, but only with specific registers.

Regarding the Virtex 5 devices, the CRC computation is done on 32 bits data width and use the same polynomial than the Ethernet CRC32 *(IEEE 802.3)* [Xilinx 2001]. To design the hardware module responsible for the CRC computation, we relied on The Virtex 5 SelectMAP simulator provided in [Xilinx 2009a]. Table B.1 shows which registers among the ICAP ones are used to perform the CRC computation.

B.3 Hardware CRC module

The hardware CRC module has an input for the current CRC value, which in our case is initialized to 0. It also has an input for the 32-bits data and the register address on 5 bits at which it has to be written. So, the algorithm implementation

Register	Address	Used
CRC	00000	No
FAR	00001	Yes
FDRI	00010	Yes
FDRO	00011	No
CMD	00100	Yes
CTL0	00101	Yes
MASK	00110	Yes
STAT	00111	No
LOUT	01000	No
COR0	01001	Yes
MFWR	01010	Yes
CBC	01011	Yes
IDCODE	01100	Yes
AXSS	01101	Yes
COR1	01110	Yes
CSB0	01111	Yes
WBSTAR	10000	Yes
TIMER	10001	Yes
BOOTSTS	10110	No
CTL1	11000	Yes

Table B.1: ICAP register involved in CRC computation

takes an input data on 37 bits as an input and return the new CRC value on 32 bits *(Fig. B.1)*.

It should be notice that the register address is not used to segregate which data should be a part of the CRC computation but as an integral part of the input data.

Table B.2 gives the resources usage of the CRC module. The Hw CRC IP is the combinatorial IP which computes the CRC. The Hw CRC PLB module is the Hw CRC IP encapsulated with a PLB bus wrapper.

Component	***Reg.***	***LUTs***	***BRAMs / FIFOs***	***DSP***	***Freq. (MHz)***
Hw CRC IP	0	160	0	0	Comb.
Hw CRC PLB	290	339	0	0	310.627

Table B.2: HW CRC Resources usage

The reconfiguration latency depends on the partial bitstream size. This size depends itself on the size of the dynamic part of the Hardware Thread. The com-

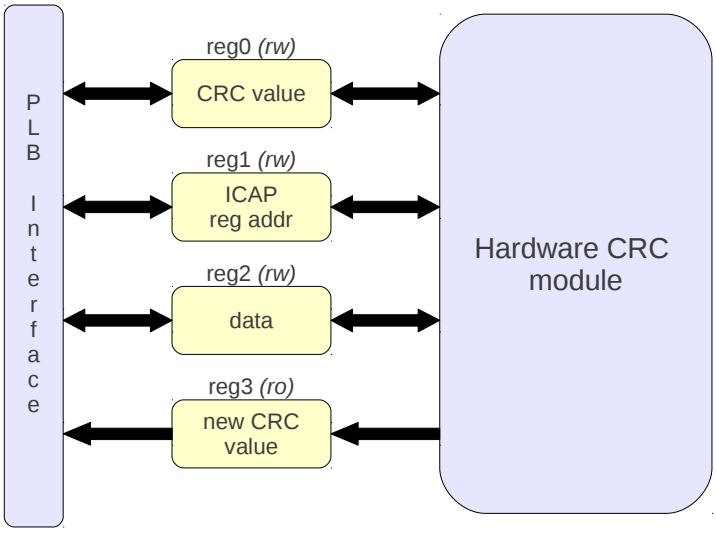


Figure B.1: CRC Bitstream Computer module

position of the Hardware Thread as well as the resources overhead caused by the encapsulation of the hardware IP is described in the next section.

Bibliography

[541 2010] *IEEE Standard for IP-XACT, Standard Structure for Packaging, Integrating, and Reusing IP within Tools Flows*. IEEE Std 1685-2009, pages C1 –360, 18 2010. (Cited on page 94.)

[Accetta 1986] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian and Michael Young. *Mach: A New Kernel Foundation for UNIX Development*. pages 93–112, 1986. (Cited on page 83.)

[Agron 2009a] J. Agron and D. Andrews. *Hardware Microkernels for Heterogeneous Manycore Systems*. In Proceedings of the International Conference on Parallel Processing Workshops (ICPPW '09), pages 19–26, Vienna, Austria, sept. 2009. IEEE. (Cited on pages 13, 20 and 21.)

[Agron 2009b] J. Agron and D. Andrews. *Hardware Microkernels for Heterogeneous Manycore Systems*. In Parallel Processing Workshops, 2009. ICPPW '09. International Conference on, pages 19 –26, september 2009. (Cited on pages 15, 94 and 95.)

[Association 2012] Multicore Association. *Multicore Association website*. `http://www.multicore-association.org/home.php`, 2012. (Cited on pages 85 and 114.)

[Baumann 2009] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach and Akhilesh Singhania. *The multikernel: a new OS architecture for scalable multicore systems*. In Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles, SOSP '09, pages 29–44, New York, NY, USA, 2009. ACM. (Cited on pages 14, 87 and 88.)

[Beckhoff 2012] Christian Beckhoff, Dirk Koch and Torresen Jim. *GoAhead: A Partial Reconfiguration Framework*. In 20th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM), pages 37–44. IEEE, 2012. (Not cited.)

[Belaid 2009] I. Belaid, F. Muller and M. Benjemaa. *Off-line placement of hardware tasks on FPGA*. In Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on, pages 591 –595, aug. 2009. (Cited on page 59.)

[Bergmann 2003] N.W. Bergmann and J. Williams. *The Egret platform for reconfigurable system on chip*. In Field-Programmable Technology (FPT), 2003. Proceedings. 2003 IEEE International Conference on, pages 340 – 343, dec. 2003. (Cited on pages 13 and 12.)

[Board 2012] OpenMP Architecture Review Board. *OpenMP website.* http://openmp.org/wp/, 2012. (Cited on page 92.)

[Bonamy 2012] R. Bonamy, Hung-Manh Pham, S. Pillement and D. Chillet. *UPaRC, Ultra-fast power-aware reconfiguration controller.* In Design, Automation Test in Europe Conference Exhibition (DATE), 2012, pages 1373 –1378, march 2012. (Cited on pages 14, 47, 48 and 59.)

[Cheng 1995] Yizong Cheng. *Mean shift, mode seeking, and clustering.* IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 17, no. 8, pages 790 –799, aug 1995. (Cited on page 136.)

[Corbett 2012] John D. Corbett. *Xilinx White Paper 412: The Xilinx Isolation Design Flow for Fault-Tolerant Systems*, January 2012. (Cited on pages 14, 50 and 77.)

[Devaux 2009] L. Devaux, D. Chillet, S. Pillement and D. Demigny. *Flexible communication support for dynamically reconfigurable FPGAS.* In Proceeding of the 5th Conference on Southern Programmable Logic (SPL 2009), pages 65–70, 1-3 2009. (Cited on page 34.)

[Donato 2005] A. Donato, F. Ferrandi, M. Santambrogio and D. Sciuto. *Operating system support for dynamically reconfigurable SoC architectures.* In SOC Conference, 2005. Proceedings. IEEE International, pages 233 –238, sept. 2005. (Cited on pages 12 and 13.)

[Duhem 2011] François Duhem, Fabrice Muller and Philippe Lorenzini. *FaRM: fast reconfiguration manager for reducing reconfiguration time overhead on FPGA.* In Proceedings of the 7th international conference on Reconfigurable computing: architectures, tools and applications, ARC'11, pages 253–260, Berlin, Heidelberg, 2011. Springer-Verlag. (Cited on pages 14, 46, 47 and 59.)

[El-Araby 2008] E. El-Araby, I. Gonzalez and T. El-Ghazawi. *Virtualizing and sharing reconfigurable resources in High-Performance Reconfigurable Computing systems.* In High-Performance Reconfigurable Computing Technology and Applications, 2008. HPRCTA 2008. Second International Workshop on, pages 1 –8, nov. 2008. (Cited on pages 13, 17 and 18.)

[Götz 2009] Marcelo Götz, Achim Rettberg, Carlos Eduardo Pereira and Franz J. Rammig. *Run-time reconfigurable RTOS for reconfigurable systems-on-chip.* J. Embedded Comput., vol. 3, no. 1, pages 39–51, January 2009. (Cited on page 95.)

[Grimm 2004] M.Ullmann M. Hübner B. Grimm and J. Becker. *An FPGA Run-Time System for Dynamical On-Demand Reconfiguration.* FPL 2004 : field-programmable logic and applications - 3203 - 842-846, August 2004. (Cited on page 44.)

[Guccione 1999] Steve Guccione, Delon Levi and Prasanna Sundararajan. *JBits: Java based interface for reconfigurable computing.* 1999. (Cited on page 44.)

[Guerin 2009a] X. Guerin and F. Petrot. *A System Framework for the Design of Embedded Software Targeting Heterogeneous Multi-core SoCs.* In Application-specific Systems, Architectures and Processors, 2009. ASAP 2009. 20th IEEE International Conference on, pages 153 –160, july 2009. (Cited on pages 15, 91 and 92.)

[Guerin 2009b] X. Guerin and F. Petrot. *A System Framework for the Design of Embedded Software Targeting Heterogeneous Multi-core SoCs.* In Application-specific Systems, Architectures and Processors, 2009. ASAP 2009. 20th IEEE International Conference on, pages 153 –160, july 2009. (Not cited.)

[Hansen 2011] S.G. Hansen, D. Koch and J. Torresen. *High Speed Partial Run-Time Reconfiguration Using Enhanced ICAP Hard Macro.* In Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on, pages 174 –180, may 2011. (Cited on pages 14, 47, 48 and 59.)

[Huang 2008] Chun-Hsian Huang and Pao-Ann Hsiung. *Software-controlled dynamically swappable hardware design in partially reconfigurable systems.* EURASIP J. Embedded Syst., vol. 2008, pages 4:1–4:11, January 2008. (Cited on page 58.)

[Huerta 2008] P. Huerta, J. Castillo, C. Sanchez and J.I. Martinez. *Operating System for Symmetric Multiprocessors on FPGA.* In Reconfigurable Computing and FPGAs, 2008. ReConFig '08. International Conference on, pages 157 –162, december 2008. (Cited on pages 14, 86 and 87.)

[Instrument 2011] Texas Instrument. *OMAP 5 mobile applications platform.* `http://focus.ti.com/pdfs/wtbu/OMAP5_2011-7-13.pdfs`, July 2011. (Cited on page 4.)

[IOC 1997] *IOCTL Specification.* `http://pubs.opengroup.org/onlinepubs/7908799/xsh/ioctl.html`, 1997. (Cited on page 12.)

[J. Carver 2008] A. Forin J. Carver N. Pittman. *Relocation and Automatic Floorplanning of FPGA Partial Configuration Bit-Streams.* Microsoft Research - Technical Report MSR-TR-2008-111, August 2008. (Not cited.)

[Kaashoek 1997] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Hector M. Briceño, Russell Hunt, David Mazières, Thomas Pinckney, Robert Grimm, John Jannotti and Kenneth Mackenzie. *Application performance and flexibility on exokernel systems.* In Proceedings of the sixteenth ACM symposium on Operating systems principles, SOSP '97, pages 52–65, New York, NY, USA, 1997. ACM. (Not cited.)

[Kallam 2009] A. Sudarsanam R. Kallam and A. Dasu. *PRR-PRR Dynamic Relocation.* IEEE Computer Architecture Letters - vol. 8 (2), September 2009. (Cited on pages 14, 45 and 46.)

[Kalte 2005] H. Kalte, G. Lee, M. Porrmann and U. Ruckert. *REPLICA: A Bitstream Manipulation Filter for Module Relocation in Partial Reconfigurable Systems.* In Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International, page 151b, april 2005. (Not cited.)

[Kamppi 2011] A. Kamppi, L. Matilainen, J. Maatta, E. Salminen, T.D. Hamalainen and M. Hannikainen. *Kactus2: Environment for Embedded Product Development Using IP-XACT and MCAPI.* In Digital System Design (DSD), 2011 14th Euromicro Conference on, pages 262 –265, 31 2011-sept. 2 2011. (Cited on page 94.)

[Koch 2009] Dirk Koch, Christian Beckhoff and Jüergen Teich. *A communication architecture for complex runtime reconfigurable systems and its implementation on spartan-3 FPGAs.* In Proceeding of the ACM/SIGDA international symposium on Field programmable gate arrays, FPGA '09, pages 253–256, New York, NY, USA, 2009. ACM. (Cited on page 66.)

[Koch 2010a] Dirk Koch, Christian Beckhoff and Jim Torrison. *Fine-Grained Partial Runtime Reconfiguration on Virtex-5 FPGAs.* In Proceedings of the 2010 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM '10, pages 69–72, Washington, DC, USA, 2010. IEEE Computer Society. (Cited on pages 49 and 66.)

[Koch 2010b] Dirk Koch and Jim Torresen. *Advances and Trends in Dynamic Partial Run-time Reconfiguration.* In Dagstuhl-Seminar 10281: Dynamically Reconfigurable Architectures, page 6, Schloss Dagstuhl, Germany, July 2010. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany. (Cited on page 6.)

[Kühnle 2006] M. Hübner C. Schuck M. Kühnle and J. Becker. *New 2-Dimensional Partial Dynamic Reconfiguration Techniques for Real-time Adaptive Microelectronic Circuits.* IEEE ISVLSI, 00:6, March 2006. (Cited on page 44.)

[Lavin 2011] Christopher Lavin, Marc Padilla, Jaren Lamprecht, Philip Lundrigan, Brent Nelson and Brad Hutchings. *RapidSmith: Do-It-Yourself CAD Tools for Xilinx FPGAs.* In Proceedings of the 21th International Workshop on Field-Programmable Logic and Applications (FPL'11), September 2011. (Cited on pages 14 and 49.)

[Lee 1987] Edward A. Lee and David G. Messerschmitt. *Synchronous Data Flow.* In Proceedings of the IEEE, volume 75, pages 1235–1245, sep. 1987. (Cited on pages 8 and 18.)

[Lee 2010] Trong-Yen Lee, Che-Cheng Hu, Li-Wen Lai and Chia-Chun Tsai. *Hardware Context-Switch Methodology for Dynamically Partially Reconfigurable Systems.* J. Inf. Sci. Eng., vol. 26, no. 4, pages 1289–1305, 2010. (Cited on page 58.)

[Leon Adams 2007] Texas Instrument Leon Adams. *Choosing the right architecture for real-time signal processing designs.* http://www.ee.up.ac.za/main/_media/en/undergrad/subjects/esp411/choosing_right_architecture.pdf, June 2007. (Cited on page 3.)

[Liedtke 2001] Jochen Liedtke, Uwe Dannowski, Kevin Elphinstone, Gerd Liefländer, Espen Skoglund, Volkmar Uhlig, Christian Ceelen, Andreas Haeberlen and Marcus Völp. *The L4Ka Vision*, April 2001. (Cited on page 83.)

[Lin 2009] Yu-Hsien Lin, Chiaheng Tu, Chi-Sheng Shih and Shih-Hao Hung. *Zero-Buffer Inter-core Process Communication Protocol for Heterogeneous Multicore Platforms.* In Embedded and Real-Time Computing Systems and Applications, 2009. RTCSA '09. 15th IEEE International Conference on, pages 69 –78, august 2009. (Cited on pages 14, 87 and 88.)

[LIP6 2012] LIP6. *MutekH website.* http://www.mutekh.org/trac/mutekh, 2012. (Cited on pages 111 and 118.)

[Liu 2009] Ming Liu, W. Kuehn, Zhonghai Lu and A. Jantsch. *Run-time Partial Reconfiguration speed investigation and architectural design space exploration.* In Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on, pages 498 –502, aug. 2009. (Cited on pages 14, 46 and 47.)

[Lubbers 2008] E. Lubbers and M. Platzner. *A portable abstraction layer for hardware threads.* In Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on, pages 17 –22, sept. 2008. (Cited on pages 13, 20 and 21.)

[Lubbers 2009] E. Lubbers and M. Platzner. *Cooperative multithreading in dynamically reconfigurable systems.* In Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on, pages 551 –554, 31 2009-sept. 2 2009. (Not cited.)

[Lysaght 2006] P. Lysaght, B. Blodget, J. Mason, J. Young and B. Bridgford. *Invited Paper: Enhanced Architectures, Design Methodologies and CAD Tools for Dynamic Reconfiguration of Xilinx FPGAs.* In Field Programmable Logic and Applications, 2006. FPL '06. International Conference on, pages 1 –6, 28-30 2006. (Cited on page 59.)

[Matilainen 2011] L. Matilainen, E. Salminen, T.D. Hamalainen and M. Hannikainen. *Multicore Communications API (MCAPI) implementation on an*

FPGA multiprocessor. In Embedded Computer Systems (SAMOS), 2011 International Conference on, pages 286 –293, july 2011. (Cited on pages 15, 92 and 93.)

[Modzelewski 2009] K. Modzelewski, J. Miller, A. Belay, N. Beckmann, C. Gruenwald, D. Wentzlaff, L. Youseff and A. Agarwal. *A Unified Operating System for Clouds and Manycore: fos.* In Rapid System Prototyping, 2006. Seventeenth IEEE International Workshop on, november 2009. (Cited on pages 14 and 89.)

[Muller 2006] C. Claus F.H. Muller and W. Stechele. *Combitgen: A new approach for creating partial bitstreams in Virtex-II Pro devices.* Workshop on reconfigurable computing Proceedings (ARCS 06) - 122-131, March 2006. (Not cited.)

[Nojiri 2009] T. Nojiri, Y. Kondo, N. Irie, M. Ito, H. Sasaki and H. Maejima. *Domain Partitioning Technology for Embedded Multicore Processors.* Micro, IEEE, vol. 29, no. 6, pages 7 –17, december 2009. (Not cited.)

[Nollet 2003] V. Nollet, P. Coene, D. Verkest, S. Vernalde and R. Lauwereins. *Designing an operating system for a heterogeneous reconfigurable SoC.* In Parallel and Distributed Processing Symposium, 2003. Proceedings. International, page 7 pp., april 2003. (Cited on pages 13, 14 and 15.)

[Nordstrom 2005] S. Nordstrom, L. Lindh, L. Johansson and T. Skoglund. *Application specific real-time microkernel in hardware.* In Real Time Conference, 2005. 14th IEEE-NPSS, page 4 pp., june 2005. (Cited on page 94.)

[OMG 2006] OMG. *CORBA Components Specification - Version 4.0*, April 2006. (Cited on page 93.)

[Rana 2007] V. Rana, M. Santambrogio, D. Sciuto, B. Kettelhoit, M. Koester, M. Porrmann and U. Ruckert. *Partial Dynamic Reconfiguration in a Multi-FPGA Clustered Architecture Based on Linux.* In Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International, pages 1 –8, march 2007. (Cited on page 13.)

[Rossi 2009] D. Rossi, F. Campi, A. Deledda, C. Mucci, S. Pucillo, S. Whitty, R. Ernst, S. Chevobbe, S. Guyetant, M. Kuhnle, M. Hubner, J. Becker and W. Putzke-Roeming. *A multi-core signal processor for heterogeneous reconfigurable computing.* In System-on-Chip, 2009. SOC 2009. International Symposium on, pages 106 –109, october 2009. (Not cited.)

[RTE 1988] *RTEMS Website.* `http://www.rtems.org`, 1988. (Cited on page 28.)

[S. Corbetta M. Morandi M. Novati 2009] M. Domenico Santambrogio D. Sciuto S. Corbetta M. Morandi M. Novati and P. Spoletini. *Internal and External Bitstream Relocation for Partial Dynamic Reconfiguration.* IEEE trans-

actions on very large scale integration (VLSI) systems - vol. 17, no11, pp. 1650-1654, October 2009. (Cited on page 45.)

[Senouci 2006] B. Senouci, A. Bouchhima, F. Rousseau, F. Petrot and A. Jerraya. *Fast Prototyping of POSIX Based Applications on a Multiprocessor SoC Architecture: "Hardware-Dependent Software Oriented Approach".* In Rapid System Prototyping, 2006. Seventeenth IEEE International Workshop on, pages 69 –75, june 2006. (Cited on pages 15, 92 and 93.)

[Shiyanovskii 2009a] Y. Shiyanovskii, F. Wolff, C. Papachristou and D. Weyer. *An Adaptable Task Manager for Reconfigurable Architecture Kernels.* In Adaptive Hardware and Systems, 2009. AHS 2009. NASA/ESA Conference on, pages 132 –137, august 2009. (Cited on pages 15, 90 and 91.)

[Shiyanovskii 2009b] Y. Shiyanovskii, F. Wolff, C. Papachristou and D. Weyer. *An Adaptable Task Manager for Reconfigurable Architecture Kernels.* In Adaptive Hardware and Systems, 2009. AHS 2009. NASA/ESA Conference on, pages 132 –137, 29 2009-aug. 1 2009. (Cited on page 17.)

[Sohanghpurwala 2011] A.A. Sohanghpurwala, P. Athanas, T. Frangieh and A. Wood. *OpenPR: An Open-Source Partial-Reconfiguration Toolkit for Xilinx FPGAs.* In Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on, pages 228 –235, may 2011. (Cited on pages 14, 49 and 50.)

[Steiger 2004] C. Steiger, H. Walder and M. Platzner. *Operating Systems for Reconfigurable Embedded Platforms: Online Scheduling of Real-Time Tasks.* IEEE Trans. Comput., vol. 53, no. 11, pages 1393–1407, November 2004. (Cited on pages 13, 15 and 16.)

[Steiner 2011] Neil Steiner, Aaron Wood, Hamid Shojaei, Jacob Couch, Peter Athanas and Matthew French. *Torc: towards an open-source tool flow.* In Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays, FPGA '11, pages 41–44, New York, NY, USA, 2011. ACM. (Cited on page 69.)

[T. Becker 2007] W. Luk T. Becker and P.Y.K. Cheung. *Enhancing Relocatability of Partial Bitstreams for Run-Time Reconfiguration.* IEEE Field-Programmable Custom Computing Machines, 2007 - 35-44, April 2007. (Cited on page 45.)

[Tanenbaum 2001] Andrew S. Tanenbaum. Modern operating systems. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2nd édition, 2001. (Cited on page 22.)

[Tomiyama 2008] H. Tomiyama, S. Honda and H. Takada. *Real-time operating systems for multicore embedded systems.* In SoC Design Conference, 2008.

ISOCC '08. International, volume 01, pages I–62 –I–67, november 2008. (Cited on pages 14 and 86.)

[Verdoscia 1994] Lorenzo Verdoscia and Roberto Vaccaro. *Actor Hardware Design For Static Dataflow Model.* In Workshop on Massive Parallelism: Hardware, Software, and Applications, pages 421–430, 1994. (Cited on pages 13, 18 and 19.)

[Wigley 2001] G. Wigley and D. Kearney. *The first real operating system for reconfigurable computers.* In Computer Systems Architecture Conference, 2001. ACSAC 2001. Proceedings. 6th Australasian, pages 130 –137, 2001. (Cited on page 16.)

[Xilinx 2001] Xilinx. *IEEE 802.3 Cyclic Redundancy Check.* Xilinx website, March 2001. (Cited on page 157.)

[Xilinx 2006] Xilinx. *Virtex FPGA Series Configuration and Readback.* Xilinx website, March 2006. (Cited on page 157.)

[Xilinx 2009a] Xilinx. *Synthesis and Simulation Design Guide.* Xilinx website, December 2009. (Cited on page 157.)

[Xilinx 2009b] Xilinx. *Virtex-5 FPGA Configuration User Guide.* Xilinx website, August 2009. (Cited on pages 14, 53, 54 and 55.)

[Xilinx 2009c] Xilinx. *Virtex-5 FPGA User Guide.* www.xilinx.com/support/documentation/user_guides/ug190.pdf, November 2009. (Cited on pages 14 and 51.)

[Xilinx 2010a] Xilinx. *Partial Reconfiguration User Guide.* http://www.xilinx.com/support/documentation/sw_manuals/xilinx12_1/ug702.pdf, May 2010. (Cited on pages 13 and 2.)

[Xilinx 2010b] Xilinx. *PLBV46 Master Burst Documentation.* http://www.xilinx.com/support/documentation/ip_documentation/plbv46_master_burst.pdf, December 2010. (Cited on page 128.)