# ARCHITECTURE OF A PROGRAMMABLE SYSTEM-ON-CHIP PLATFORM FOR FLEXIBLE RADIO PROCESSING

**BY ONKAR SARODE**

**A thesis submitted to the**

**Graduate School—New Brunswick**

**Rutgers, The State University of New Jersey**

**in partial fulfillment of the requirements**

**for the degree of**

**Master of Science**

**Graduate Program in Electrical and Computer Engineering**

**Written under the direction of**

**Prof. Predrag Spasojević**

**and approved by**

_____

_____

_____

_____

**New Brunswick, New Jersey**

**October, 2010**

## ABSTRACT OF THE THESIS

# Architecture of a Programmable System-on-Chip Platform for Flexible Radio Processing

**by Onkar Sarode**

**Thesis Director: Prof. Predrag Spasojević**

The emergence of multiple *radio access technologies* (RATs) and their continuous evolution, is driving the need for programmable radio processing. Programmable radio devices with run-time flexibility and resource virtualization features will not only enable faster *time-to-market*, longer lifetime of devices, and universal connectivity, but also act as building blocks for advanced wireless technologies of adaptive and cognitive radios. These requirements have forced a shift from the traditional ASIC approach. However, most existing flexible solutions are based on either fully software-defined or software-controlled approaches that lack the power efficiency, performance and determinism (for real-time constraints) needed for wireless processing.

In this thesis, we propose a programmable multi-processor *system-on-chip* (SoC) platform architecture based on a novel *Virtual Flow Pipelining* (VFP) framework that aims at striking a balance between flexibility (as provided by SDR) and performance (as provided by ASICs). The key highlights of this concept are a simple task-level programming model for provisioning protocol flows, and the use of dedicated hardware-based OS-like support for controlling their run-time execution. We present the evolution of

a clustering-based organization for the SoC with distributed-shared controllers. Clustering along with an inherent architectural support for message passing provides a balance between scalability and hardware overhead. Shared controllers with a pipelined microarchitecture and a separate interconnect for control messaging are designed for low hardware complexity and high performance.

The proposed architecture is evaluated by creating a bit- and cycle-accurate model in synthesizable *register-transfer-level* (RTL). It has been built into a virtual platform for 802.11a transmitter, which has successfully executed single and multiple flows for rates of 6, 12 and 24 Mbps. This thesis also presents a characterization and analysis of the architecture to provide key implications such as control overhead for different task sizes, its impact on cluster size etc.

# Acknowledgements

Foremost, I would like to express my sincere gratitude to my advisors, Prof. Zoran Miljanić and Prof. Predrag Spasojević, for their continuous support and guidance throughout my thesis. I am indebted to Prof. Miljanić for his confidence in my abilities and all the opportunities he has provided. I cannot thank him enough for the great learning experience that this has been. I am grateful to Prof. Spasojević for giving me complete freedom in my work and teaching me to think with respect to the *big picture*.

I would also like to thank Khanh Le for all the *anytime* discussions and brainstorming sessions that have taught me many detailed aspects of digital design. Khanh's modesty and helpful nature are an inspiration. Many thanks to Ivan Seskar for his timely help and support.

Working at WINLAB would not have been half as enjoyable without my friends; Akshay, Mohit, Madhura, and Prashant. A special thanks to my roommates; Arjun Adimari, Pandeyji, Pompy, and Srini, for adding the fun quotient to my life at Rutgers.

Most importantly, I would like to thank my family. I cannot even imagine pursuing my dreams and ambitions without the unconditional love, support, and encouragement of my parents; Mrs. Manjiri Sarode and Mr. Anant Sarode. I owe my every success to them. Many thanks to my brother, Swatantra, for taking up my share of responsibility back at home. Last but not least, I would like to thank my fiancée, Poonam Deshmukh, for her patience and sacrifices, and always being there for me.

# Table of Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

Evolution of wireless devices is at a very exciting stage. We are now amidst of an accelerating trend towards intelligent devices, such as smart phones, that converge services and functionality earlier provided by cell phones, tablet computers, MP3 players etc. Maximizing the efficacy of these devices calls for providing wireless connectivity across space and time. In fact, applications in the near future will demand availability of ubiquitous connectivity in a seamless manner. However, providing such connectivity is difficult because there exists no single universal wireless technology that covers the entire world. This recent and projected growth in the demand for wireless connectivity together with the emergence of diverse *radio access technologies* (RATs) and standards, has opened significant opportunities and challenges in the wireless communication devices industry.

Future wireless devices must incorporate multiple communication functions provided via different standards, e.g. personal navigation (GPS), personal area networking (Bluetooth, Zigbee etc.), local area networking (802.11a/g/n), TV reception (MediaFLO, DVB-H/T etc.), and mobile cellular networking (LTE, WiMax) on a single device. With this increase in the ways and means by which people need to communicate − i.e. data communications, voice communications, video communications, broadcast messaging, command and control communications, emergency response communications, etc. − modifying radio devices easily and cost-effectively has become *business critical*. The emerging 4G wireless standards itself impose heterogeneous wireless communication environments where the infrastructure will be built with devices using different radio access technologies, and operating at different spectrum bandwidth. Furthermore, the cognitive radio technology, seen as the solution to deal with the problem of spectrum

scarcity, demands broader flexibility and agility features from the wireless devices.

## 1.1 Software Defined Radio

Software defined radio (SDR) [1] technology is seen as the solution for providing the flexibility, cost efficiency and power to drive communications forward, with wide-reaching benefits realized by service providers and product developers through to end users. Here we shall define a Software Defined Radio and discuss the factors/benefits that drive this technology. Understanding these driving factors is vital to be able to appreciate the implications they have on the requirements/specifications for these devices (discussed in Chapter 2).

Simply put, a SDR is defined[2] as a *Radio in which some or all of the physical layer functions are Software Defined*; where Software Defined refers to *the use of software processing within the radio system or device to implement operating (but not control) functions.* This definition implies that the complete physical layer processing for the SDR is performed by software routines running on processor(s). A later discussion will explain how our programmable solution contrasts with the SDR notion. But, for now we put forth the driving factors for the SDR because these are the very same factors that drive a programmable system-on-chip (SoC) solution that we propose.

## 1.2 Factors Driving the SDR

The factors driving the SDR can be divided into two broad categories: *Short Term drivers* and *Long Term drivers.* Short Term drivers refer to the factors that influence the adoption of this technology in the near future, i.e., 5 years or so. On the other hand, Long Term refers to the factors and visions, related to the advanced wireless market in the distant future, for which the SDR can act as a key enabling technology.

### 1.2.1 Short Term Driving Factors

Benefits of SDR will be seen throughout the wireless industry chain − from product-based and service-based providers through end users of the devices and services.

**Factors affecting network operators**

Network operators have to deal with the constant evolution of standards and technologies. For example[3], LTE Advanced is currently in the works and it features a full OFDMA uplink physical layer as opposed to SC-FDMA for LTE. Such evolutions take place over a short period of time − sometimes as low as two years − which is short in comparison to the time it takes to develop, test and validate equipment.

For network operators, SDR enables new features and capabilities to be added to existing infrastructure − increasing the *time-in-market* for their purchased equipment. This allows network operators to quasi-future proof their networks − saving major new capital expenditures. The ease of upgrades means faster deployment of new standards and services. The use of a common radio platform for multiple markets, significantly reducing logistical support and operating expenditures.

**Factors affecting product-based providers**

SDR approach implies the use of a common platform for implementation of diverse wireless technologies and standards. The impact of such a platform-based approach is that it will dramatically reduce the *time-to-market* for new products. By facilitating extensive reuse, this approach will also reduce the effort involved in development and testing of new products, thus lowering the costs. Thus, infrastructure equipment vendors, system integrators and terminal device makers can all roll out "family" of radio products e.g. macrocells, femtocells etc. in a fast and cost effective manner. New algorithms and software will be targeted for a proven framework, thus making the process of estimating their performance, latency etc. not only easier but also more deterministic. It also makes debugging and upgrading devices easy.

Furthermore for terminal device manufacturers, SDR also acts as a enabling technology towards the need for a cost-effective multi-modal solution − one that can provide connectivity across heterogeneous RATs. The benefit of providing the ability to evolve/upgrade with improvements in standards applies here as well.

#### Factors affecting end-users

For end-users, the SDR offers ubiquitous wireless connectivity – i.e. the means to connect using whatever means (RATs, bands etc.) available. The easy-to-upgrade feature of SDRs imply a longer lifetime for the mobile devices – thus reducing costs.

### 1.2.2 Long Term Driving Factors

Software Defined Radios are seen as the potential building blocks for advanced wireless technologies – *Adaptive Radio, Cognitive Radio*[1, 4] – which act as a major driving factor for SDR technology from a long term perspective.

The next couple of paragraphs very briefly introduce the advanced adaptive and cognitive wireless technologies, which themselves are vast topics of very active research.

#### Advanced Wireless Technologies – Adaptive and Cognitive Radios

Adaptive radio is a technology in which communications systems have a means of monitoring their own performance and modifying their operating parameters to improve this performance. The use of SDR technologies in an adaptive radio system enables greater degrees of freedom in adaptation, and thus higher levels of performance and better quality of service in a communications link.

Furthermore, building on adaptive radios, the cognitive radio technology[5, 1] offers to solve the problem of efficient utilization of the radio spectrum, which has become critical in order to accommodate the exponential growth of wireless devices in the future. These future cognitive radios will perform continuous or frequent sensing of their network environment (spectrum, interference conditions etc.) and dynamically change (their RAT, operating frequency, power control etc.) to make optimum use of the available spectrum while providing seamless connectivity.

#### SDR – Key Enabler for Advanced Wireless Technologies

In order to accomplish these advanced features, we need wireless devices that can adapt to a variety of radio interference conditions and protocol standards. Such a radio should

be capable of dynamic physical layer adaptation via scanning of available spectrum, selection from a wide range of operating frequencies, rapid adjustment of modulation waveforms and adaptive power control.

SDR technology has the capability of providing the physical layer flexibility required for realizing these technologies. Thus adoption of the SDR technology is critical in allowing end-users to make optimal use of available frequency spectrum and wireless networks, with a common set of radio hardware.

Having discussed the positive benefits driving the SDR concept, we now introduce the challenges and difficulties involved in realizing this technology. From hardware design point of view, the challenges are enormous, both on the *radio-frequency* (RF) and analog front-end and on the digital baseband side. This thesis focusses on and discusses only the digital baseband challenges and solutions. The following section is aimed at discussing, from a high-level perspective, the difficulties of adopting the SDR approach as well as at presenting an overview that can act as a platform for introducing our work. More details of related work, their aspects and contrast with our solution will be covered in Chapter2 after we explain the specifications/metrics for comparison.

## 1.3 True SDR versus ASIC: the flexibility-performance gap

The discussion up to this point focussed only on the features, flexibility and driving factors related to future wireless devices. With this myopic view, fully programmable SDRs seem to be the dream solution. In fact SDR has been a dream for the last decade – without any solutions available in the volume market. The major gap between this *dream* (SDR) and *reality* (traditional ASICs) is performance and power.

As we have already seen, the need for adaptability, reduced *time-to-market*, longer *time-in-market* etc. all point against the full custom ASIC solution. But the ASIC approach is capable of delivering on the low-cost, performance and power requirements of future wireless protocols. Resorting to a technique of building multi-modal radios

comprising of multiple ASIC modules (one for each standard), is not a feasible solution. The current requirement itself will demand at least 16 baseband modules[6], which will result in heavy silicon and prohibitive costs, let alone the incapacity to evolve with the volatile standards.

Fully software programmable solutions built using CPUs − true SDR − depicted in Fig.1.1a, although providing maximum flexibility, are compute centric. They are not suitable for delivering the requirements of the network-centric world − i.e.[7] a short sequence of data manipulation operations, sequential nature with high data dependency. Moreover, these solutions are extremely power hungry.

## 1.4 Bridging the gap: a hardware-oriented, programmable SoC solution

To keep the cost low to enable a deployment model that can provide the capacity for future data traffic requirements while allowing evolution with wireless standards creates a dichotomy that can be resolved through a new generation of *platform-based System-on-Chip* (SoC) devices that combine aspects of programmable devices and ASICs in a balanced manner.

A key aspect to consider is that providing wireless devices with flexibility *within certain reasonable bounds* is enough. A fully programmable solution is an overkill. Metaphorically, if the problem is undoing a screw, and the type of screw head is unknown, then carrying a single screwdriver with modifiable screw-ends is enough. Carrying a complete toolset is not required. A solution that provides limited versatility − to support diverse RATs and adapt to their evolutions − is enough.

Depicted in Fig.1.1b, the *Software Controlled Radio* [2] concept, i.e., using a mix of *parameterizable hardware functional units* (FUs) and *application specific instruction set processors* (ASIPs) under the control of a central software programmable CPU, is a step in right direction for achieving the required balance. Nevertheless, for reasons discussed in Section 2.2, the software controlled approach cannot deliver the performance and power requirements. This approach proves even weaker, when we consider support

for virtualized multiple traffic flows etc., which are required by the advanced wireless technologies.

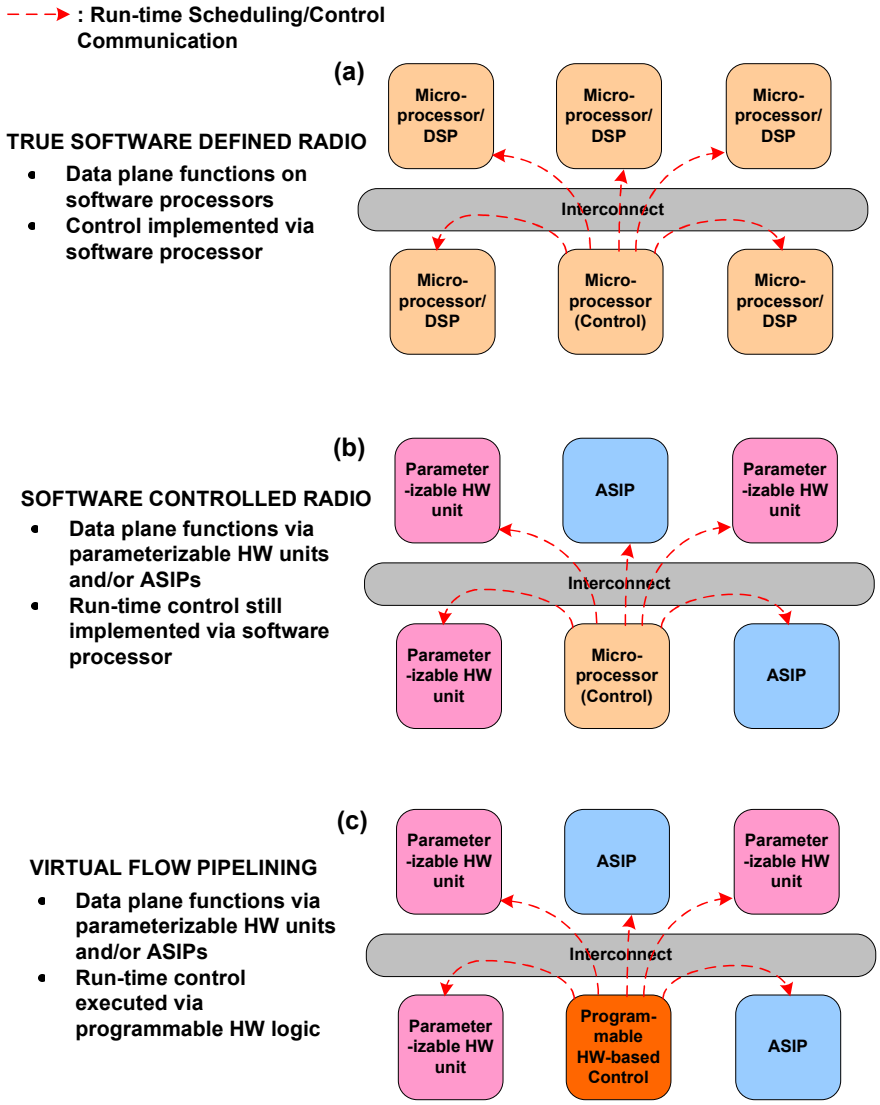


Figure 1.1: SDR v/s SCR v/s VFP

### 1.4.1 Introducing Our Solution – Background and Overview of the Thesis

To summarize the implementation challenges, they can be broadly categorized under RF and Analog Front-end design; and Digital Baseband Design. As mentioned previously, our work focusses on the Digital Baseband part. In fact, the Digital Baseband Design

challenges can be further classified as;

1. *related to processing/functional units*: These deal with the design of the data plane processors/functional units. The issues here involve analysis and exploration of algorithms, their hardware/software partitioning and implementation, making them parameterizable and flexible etc.

2. *related to system integration, programmability and run-time control*: This deals with the challenges in creating a framework, which provides all the required mechanisms for programmability, scheduling of the numerous different tasks running on very different operating units and executing the program within a real-time constraint, while accurately resolving the dependencies and exploiting parallelization possibilities. This challenge is precisely the the scope of our work.

This thesis proposes a programmable SoC architecture, based on a novel *Virtual Flow Pipelining* (VFP)[7] framework (elaborated in Chapter3), wherein the system control, while being programmable, is implemented as dedicated hardware − not using a central software programmable CPU with a real time operating system. Fig.1.1c depicts the concept. We refrain from calling our solution a SDR or SCR, which implies a software programmable CPU based control. Instead we have OS-like hardware based support that executes *soft* control flow programs. We aim at striking a favorable balance between flexibility and performance. An important point is that the VFP based approach targets better performance and power than the Software Controlled approaches, while providing the required flexibility.

The architecture proposed in this work is a clustering-based architecture which aims at striking a balance between fully distributed and fully centralized approaches. The architecture uses distributed-shared controllers with a scheme for message passing to handle scalability. This enables us to achieve scalability with a reasonable hardware overhead. Another major aspect, of this work has been the architecture and design of the shared controller. The challenge was to enable the controller to support maximum number of processing elements with minimum impact on performance. This has been achieved by designing an asynchronous pipelined microarchitecture for the controller

that can simultaneously serve multiple processing elements. Identifying the parallelism in the VFP control mechanisms and exploiting it using the pipelined architecture is a major contribution of this work. Furthermore, a new interconnect for control communication was designed to improve performance. The concept of IDs is introduced and uniform messaging schemes/formats and interfaces for the processing elements are designed for communication with the VFP controller. This work has implemented the complete system in synthesizable *hardware description language* (HDL), enabling bit and cycle accurate simulations and performance analysis. During the coarse of this work, 802.11a-like OFDM based baseband processing has been successfully implemented on the system[8]. More details will be covered in later chapters.

The organization of the rest of the thesis is as follows; based on the driving factors discussed in this chapter, Chapter 2 will discuss the specific requirements/specifications of the programmable radio devices. Using these specifications as criteria, Chapter 2 will also discuss some of the existing and related work/solutions as well as compare and contrast them with our solution. Chapter 3 will describe the Virtual Flow Pipelining architectural framework and programming model. Chapter 4 will delve into the architecture of the SoC and controller. Experimentation and performance analysis is covered in Chapter 5. Finally, Chapter 6 concludes with the insights and scope for future work.

# Chapter 2

# Requirements of Flexible Baseband Processing for Cognitive Radio Applications and their Implications

In Chapter 1 we discussed the factors driving the need for flexible solutions for the wireless radio processing. We also introduced the challenges involved in realizing such a solution, namely the performance-flexibility gap between the established ASIC approach and the desired SDR approach. This chapter will identify and present the specific requirements of a flexible solution. The requirements presented in this chapter can be thought of as higher level specifications for the architecture and will also act as guidelines for drawing comparisons with other related work.

As mentioned in the previous chapter, this work focuses only on the digital baseband challenge of the radio system. More specifically, we concentrate on the system integration, programmability and run-time control/scheduling of the digital baseband. Hence, the next section will focus more on the framework requirements rather than those of constituent data-plane processors and/or hardware functional units. Later in 2.2 we discuss the impact of these requirements on the architectural decisions.

## 2.1 The Requirements

Before, we begin elaborating the requirements it is important to mention that, from an application point of view, we are targeting a system that together with the short term SDR factors, can also support the cognitive and adaptive radio applicative scenarios (long term factors of Chapter 1). Broadly speaking the goal is to create a *Dynamically Reconfigurable Platform*[9]. The following sections put forth the requirements and their justification for such a system.

### 2.1.1 Platform-based Paradigm

The realization of the flexible wireless digital basebands calls for an evolution from the *Intellectual Property* (IP) reuse paradigm to a platform reuse paradigm[9]. Simply putting, a platform is a SoC communication structure on which different IP can be seamlessly connected. [10] defines a platform as a *layer of abstraction with two views*. The upper view allows an application to be developed without referring to the lower levels of abstraction. The lower view is a set of rules that classify a set of components belonging to the platform.

The main requirements that a platform targeting digital baseband must provide are:

1. A network-centric framework that provides separation between communication and computation aspects of the executed protocols. The communication scheme architected must be independent of the type of interconnect used (bus, crossbar, etc.), thus providing support for the *Globally Asynchronous Locally Synchronous* (GALS) approach, which is now imperative for seamless IP integration in large SoCs.

2. A programming model or *Application Programming Interface* (API) which is a high level interface to the hardware. This provides the required upper view or level of abstraction for easily programming the system. This decouples the programming aspect of the platform from the microarchitecture aspects as well as makes the programs independent of the actual functional units or processors used for executing the data operations.

3. Integration of heterogeneous processing engines within unified programming paradigm. This is even more important because there is a consensus appearing[9][11][12][13][14], which states that the way of solving the tradeoffs between power, performance and flexibility is the use of *heterogeneous multi-processor SoC* (MPSoC) based solutions (discussed in section 2.2). Therefore, a platform should provide the uniform rules/interface to seamlessly integrate these heterogeneous computing cores. This will help create a library of components that are compliant with the platform, thereby increasing their reusability and making system integration faster

and easier. Development and/or enhancements of the IP components can be done completely independently and in parallel − provided the platform/interface rules are obeyed.

4. System integration and verification phases becomes straightforward. IPs can be developed independently without caring about communication with other IPs. This boosts the reusability of cores and enable developing families of products. Thus, reducing the time to market and *non-recurring engineering* (NRE) costs.

5. Programmability is facilitated by the common platform programming model or API. This makes mapping of protocols onto the hardware easy and reconfigurable. Thus, a platform-based approach lends itself to the realization of a flexible solution. It also enhances the software reusability and makes upgrades easy.

### 2.1.2 Programmability/Configurability

We have already mentioned that there is a growing demand by consumers for connectivity everywhere. This demand, together with the goals of small size, low cost, and power efficiency are driving the need for multi-modal solutions that are implemented as a common configurable hardware platform (as opposed to having independent hardware per RAT). In fact this is the very basic idea behind the concept of SDR.

Furthermore, there is practically no doubt that the MPSoC ([9][11][15][13][16][14]) is a promising approach to solve the challenge of improving the contradicting objectives of performance, power efficiency and flexibility.

We must be able to program the MPSoC in order to execute a complete wireless communication protocol. Moreover, changing the configuration/program must enable the implementation of different protocols on the same platform. The following points explicitly define the properties of the required programmability/configurability and degrees of freedom, which are required to be provided by the platform architecture in order to support diverse RATs.

**Configurability of the Computing Cores:** This degree of freedom primarily deals with the computational aspect of the platform. The MPSoC approach involves a

new step in programming of the chip, where constituent computations/functions of the protocol must be identified, extracted and mapped onto the constituent processing elements (PE). Thus the different PEs that make up the MPSoC must be configurable computing cores − i.e. parameterizable hardware functions for computational intensive tasks that need less flexibility (e.g. FFT) or general purpose RISC processors or application specific instruction set processors (ASIP). The platform programming model must provide uniform way − irrespective of the type of PE − for selecting the required configuration/function of the PE, as per the protocol under execution. The nature of this programmability/configuration depends on the granularity of PEs − coarse-grain or fine-grain. For example, while executing a WiMAX transmitter on a MPSoC with coarse-grain PEs, the interleaver ASIP must be configured to perform the WiMAX interleaving algorithm. This configuration must be changed to run the 802.11a interleaver algorithm when the device has to transmit 802.11a packets.

**Configurability/Programmability of the Control-Flow:** The platform must have a programmable control-flow as per the protocol being implemented. This degree of freedom deals with the programmable nature of the sequence of the constituent computations/functions of the protocol that being implemented on the PEs. The programming model must provide flexibility in defining the producer-consumer relations, which must be supported by the hardware platform in the form of configurable communication structure.

**Freedom to Upgrade/add FU Configurations:** This degree of freedom enhances the ability of the platform to evolve with emerging protocols, specifically when it needs a completely new computational task not considered at design time. This aspect applies only to the programmable processor-based components of the MPSoC, where new programs on the CPUs will manifest as new configurations. A thorough analysis of the value and affordability of the degree of programmability required for different functions needs to be performed. For example, a configurable hardware engine capable of multi-length (over a wide range) multi-stream FFT

core is enough − dedicating a CPU for the FFT function is neither affordable nor does it add any value.

All these above mentioned of degrees of freedom, together with the ability to exploit task-level parallelism, must be built into the programming model.

### 2.1.3 Dynamic Reconfigurability

The technologies of Adaptive Radio and Cognitive Radio take the programmability/configurability requirement to the next level. The solutions for efficient spectrum usage need the wireless devices to dynamically reconfigure based on the changes in the radio environment. In fact, this agility in adaptation is required on a per-packet basis.

This requirement of *dynamic reconfigurability at run-time on a per-packet basis* is also driven by the user demand for seamless connectivity when moving across heterogeneous RATs. In fact this requirement for seamless connectivity has even more implication on the requirements, discussed in the next section.

### 2.1.4 Multiple Simultaneous Traffic Flows

Consider the following applicative scenario: A person starts a video conference with a friend while leaving office in a business center, then takes the train to go home in the suburbs. During the person's journey the video conference must be continued without any disconnections. This demand of seamless and ubiquitous connectivity for users − during the course of an active application − needs to be supported by *soft vertical handovers* across the diverse RATs that are available. The implications of this on the digital baseband platform is that it needs to support at least 2 diverse RATs simultaneously [9][17].

In fact users will be running multiple radio applications simultaneously on the same wireless device − e.g. GPS and Bluetooth file transfer to a friends phone together with data download from the internet. Such scenarios drive the need for more than 2 simultaneous diverse traffic flows on the platform. This support for independent radio links is also needed by the future cognitive radios in order to communicate (control and

data) and co-operate with multiple radio neighbors for opportunistic spectrum usage as well as formation of ad hoc networks.

### 2.1.5 Resource Virtualization and QoS guarantees

Satisfying the previous two requirements − simultaneous support for multiple traffic streams and dynamic (per-packet) reconfigurability − with lowest cost requires sharing the hardware resources of the platform. At the same time every radio session is also required to adhere its relevant real-time constraints. This poses a difficult system problem − as satisfaction of temporal constraints depends on the resource availability, resource sharing can make the temporal behavior of each radio session (or traffic flow) depend on the behavior of all other simultaneous radio sessions in the system, which is difficult to predict since the combinations of radio sessions change dynamically.

This drives the need for virtualization support that will isolate each radio session such that it only sees a fraction of the platform resources. It will also protect the higher protocol layers from the changes in radio access features and loading on the hardware[7][18].

Thus the platform API layer or programming model will have to incorporate virtualization features. Primarily, this involves handling the hardware processing resource allocation for enforcing the QoS guarantees across the multiple traffic sessions while maintaining isolation between them. Thus from the point of view of higher layers, each session treats its share of bandwidth as a separate (virtual) channel that is unaware of other sessions in the system.

### 2.1.6 Throughput, Real Time Constraints and Power Efficiency

The throughput requirements of digital baseband processing for $4G$ devices are in the range of $100Mbps$ data rates which translates to approximately 500 giga operations per second (GOPS)[11]. Moreover, this high throughput must be delivered within a minimal power budget of around $200mW$[11][19].

In addition to the throughput requirements, most of the processing is bound by tight real-time constraints; e.g. WLAN packet retransmission has strict deadline (SIFS time)

that is 16 micro seconds, which involves the entire receive and transmit paths. A failure in meeting these real time constraints will cause the processing to be worthless. These type of strict latency and timescale requirements, makes the scheduling and execution of the tasks on the platform challenging.

## 2.2 Implications of the Requirements

### Heterogeneous MPSoC

In order to provide required performance within the power budget, the solution has to be MPSoC based, as multiple simpler cores running at lower frequency are more power efficient. Moreover, the functional analysis[14] of the tasks performed in the the digital baseband processing has revealed a clear diversity of computational load and flexibility requirements. Hence, a *heterogeneous MPSoC* is the best suited solution for our requirements of sections 2.1.2 and 2.1.6 − providing flexibility only where it adds value and can be afforded.

### Run-time Task Scheduling

The requirements of data dependent control-flow, dynamic (per-packet) reconfigurability (section 2.1.3), and maintenance of real-time constraints and performance guarantees across multiple radio sessions (sections 2.1.5, 2.1.6); on a heterogeneous MPSoC cannot be satisfied by compile-time scheduling. This dictates the need for run-time scheduling.

### Hardware Support for Programmability/Scheduling

Providing this run-time scheduling support is difficult as well as inefficient with a software controlled approach. The software control implies that the scheduler is implemented as software on a CPU and will require a *real time operating system* (RTOS) for sharing the CPU processing power. Running the RTOS will itself consume considerable energy. Moreover, the synchronization between the control code (running as software) and the processing elements (performing data-plane processing) will be interrupt based. To avoid high interrupt overheads, the execution time of the PEs must at least be a

degree higher (thousands of cycles) than the interrupt latencies (hundreds of cycles) of the operating system environment. As an interesting example[20], consider the interrupt latency is around 300 cycles and 50,000 interrupts are generated per second, this overhead itself swallows 10% of the $150MHz$ processor resources. Also the delayed responses to the PEs will decrease the PE utilization as well as increase system latency.

Broadly speaking, the time scales of real-time operating system slices and CPU context switching in the software controlled environment are an order of magnitude larger than the ones required by the wireless protocol processing (tens of $\mu$s vs. $\mu$s)[7]. Hence, Software Controlled Radio platforms are not adequate for the multi-flow communication support since time slicing of processing resources will be inefficient and non-precise.

These implications drive the need for a architectural framework with hardware-oriented support for dynamic task scheduling/programmability on the heterogeneous MPSoC. This architecture must have mechanisms for efficient resource sharing, low overhead context switching, high utilization of PEs and exploiting the task-level parallelism inherent in wireless protocol processing. The *Virtual Flow Pipelining* (VFP) concept is a novel architecture framework that caters to these requirements and their implications.

## 2.3 Related Work

A number of architectures and implementations for flexible baseband processing are available in literature. But none of them cover all the requirements and their implications mentioned in the prior sections. We discuss some of these below.

The work by G. Fettweis *et al* [13][21] incorporates the use of a dedicated hardware unit for programmability/scheduling with real-time support. This work is the most closely related to ours. But, unlike our scheme, they resolve their control dependencies in software. This causes stalling due to long interrupt latencies. Also they have a fully centralized approach for which the performance degrades for tasks that are smaller than 4000 clock cycle. As shown in Chapter 5, our performance is better with efficient support for short tasks. Also, their solution cannot support simultaneous multiple flows.

In fact, as described in [22] they too resort to a software approach in order to support multiple flows. The good point about their work is that it is C-programmable.

Another interesting approach is by [40], where they have their control implemented using a SW programmable processor − but the processor is an application specific processor (ASIP), designed with special hardware supported instructions for scheduling etc. This gives the approach the programmability of a SW approach with better performance. However, as will be shown in Chapter 5 the performance is limited when compared to ours. Also, our argument is that we really do not need that much programmability.

The IDROMEL platform[17] from OpenAirInterface use a software based approach, but claims to performance by delocalizing part of the scheduling to local MIPS microprocessors − present in every processing block. They aim at support for only 2 simultaneous RATs − thus lacking support for virtualization.

[23] too has a distributed software based control with a MIPS processor embedded in every processing element. Real-time scheduling is achieved by avoiding the use of an operating system and interrupt based mechanisms. Instead a polling based event detection (per control processor) is used. Also, this is a homogeneous MPSoC solution, hence probably the MIPS based software control makes sense − complex scheduling techniques can be implemented. Support for multiple flows and virtualization is missing.

IMECs COBRA [24] is perhaps one solution that meets most of the requirements described in this chapter. But not many details about it are available in literature. It targets 4G giga-bit rates and claims support for multiple simultaneous radio sessions with real-time guarantees.

The architectures of PicoChip[25], Infineon's MuSIC[26] and Sandbridge's SB3011 platform[27] argue high computational performance and high flexibility. But they are all DSP-centered and accelerator-assisted with a centralized control − reconfiguration and scheduling is performed by one central software processor. This approach is not scalable and complex control strategies (pipelining, etc.) are difficult and inefficient to implement. Also reconfiguration is often slow.

SODA[28] too has CPU based control and scheduling. It also lacks support for

dynamic reconfiguration, multiple flows and virtualization. Moreover, its performance is limited − supports maximum of 24Mbps rate for the 802.11a protocol at 400 MHz. Our platform can achieve the same performance at just 100 MHz.

# Chapter 3

# The Framework: Virtual Flow Pipelining

The previous chapter elaborated on the requirements of digital baseband platforms for supporting future wireless technologies. We also discussed the implications of the requirements on the approach to be taken towards realization of the required features. Finally, with these guidelines for reference we evaluated some of the existing solutions in literature.

This chapter will elaborate on the specifications and features of the *Virtual Flow Pipelining* (VFP) framework, which is the basis for the programmable SoC architecture. As introduced in the Section 1.4.1, the VFP architecture framework addresses the workload characteristics of wireless communication protocols with hardware-based programmable control mechanisms that engage both hardware and software modules in a uniform manner in order to satisfy both functional and performance requirements. The following sections detail the features, the programming model, and the mechanisms of the VFP framework.

## 3.1 Divide and Conquer Approach: Task Level Programmability with Coarse Grain PEs

As per the arguments of Section 2.2 and [15, 9, 11, 12, 13, 14], we have concluded that the MPSoC is the most suitable approach for achieving the flexibility and performance requirements of emerging wireless protocols within the strict power budgets. The MPSoC approach brings with it the challenge of distributing the workload across its constituent *processing elements* (PEs) and programming the system.

### Power related considerations

Before we can actually tackle programmability/scheduling related challenges we need to decide the nature of the PEs. This is an important design issue because it impacts the power efficiency of the system. Analysis ([29, 14, 30]) of the workload of wireless baseband processing has led to the realization that many of their constituent functional entities are similar − e.g. channel encoder/decoder, block interleaver/deinterleaver, modulator/demodulator, pulse shaping lter, and channel estimator. The specific algorithms that each of these entities performs − e.g. type of interleaving etc. − will depend on the standard/protocol being executed. Moreover, these different functional entities offer plenty algorithm/task level parallelism − implying each of these can be executed in parallel without intensive interactions. In fact, the analysis in [29] shows that in most computation intensive algorithms, the amount of memory access for internal computation is about 10 to 200 times greater than that for the communication with other algorithm blocks. It means that if a PE is strong enough to cover one entire signal processing algorithm, the amount of interprocess communication can be minimized.

Minimizing the amount of interprocess communication is important in a low power system because the energy cost of inter process communication (involving access to system interconnect etc.) is at least two times higher than that of internal memory access. This is the primary reason we choose to use coarse grained PEs.

### Throughput related considerations

From the view point of system throughput, the coarse grain PE is also a better choice because the communication delay between ne grain PEs degrades system throughput. Usually, the operation speed of the interprocess communication network is slower than the internal memory access. Furthermore, the interprocess communication time is not deterministic. So, the ne grain PEs must be scheduled under the assumption of worst case delay. These factors degrade achievable maximum system throughput, if fine grained PEs are used.

### Flexibility related considerations

The use of coarse grained PEs, each executing specific tasks/algorithms, paves the way for the use of heterogeneous PEs. As discussed in Section 2.2, the diversity of computational complexity and flexibility among the functional tasks/algorithms further advocate this *divide-and-conquer* approach. It facilitates achieving flexibility while using specialized cores − which help achieve power efficiency by adding flexibility and/or computational power only when required and afforded.

Thus, with all these considerations, the VFP framework uses a heterogeneous MPSoC approach with coarse-grain PEs, which can be configured to execute constituent tasks/kernels of the wireless protocol. These tasks are stitched together by another higher level of programmability, which in effect implements the intended wireless protocol.

## 3.2 What is a Virtual Flow?

The allocations of the share of platform hardware resources to a protocol flow, specifying the tasks involved, their sequence and performance requirements, in effect creates a *Virtual Flow*. Multiple such virtual flows can be supported by the VFP framework.

Consider a heterogeneous MPSoC that includes the PEs shown in Fig. 3.1a. This figure is meant to be an abstract view of the MPSoC platform, meant for building the notion of virtual flows and comparing it to the traditional hardwired pipeline. Details regarding the SoC organization, microarchitecture, interconnect etc. will be discussed in the next chapter.

Building on this abstract view, Fig. 3.1a shows two virtual flows being executed on the MPSoC. The blue path depicts virtual flow 1 ($VF1$) and the red path depicts virtual flow 2 ($VF2$). As can be seen each flow has its own sequence as well as different operations to be performed within each stage of the sequence. Also, both these flows coexist on the MPSoC platform concurrently, with their performance guarantees (QoS requirements) specified as part of the flow timings and priorities.

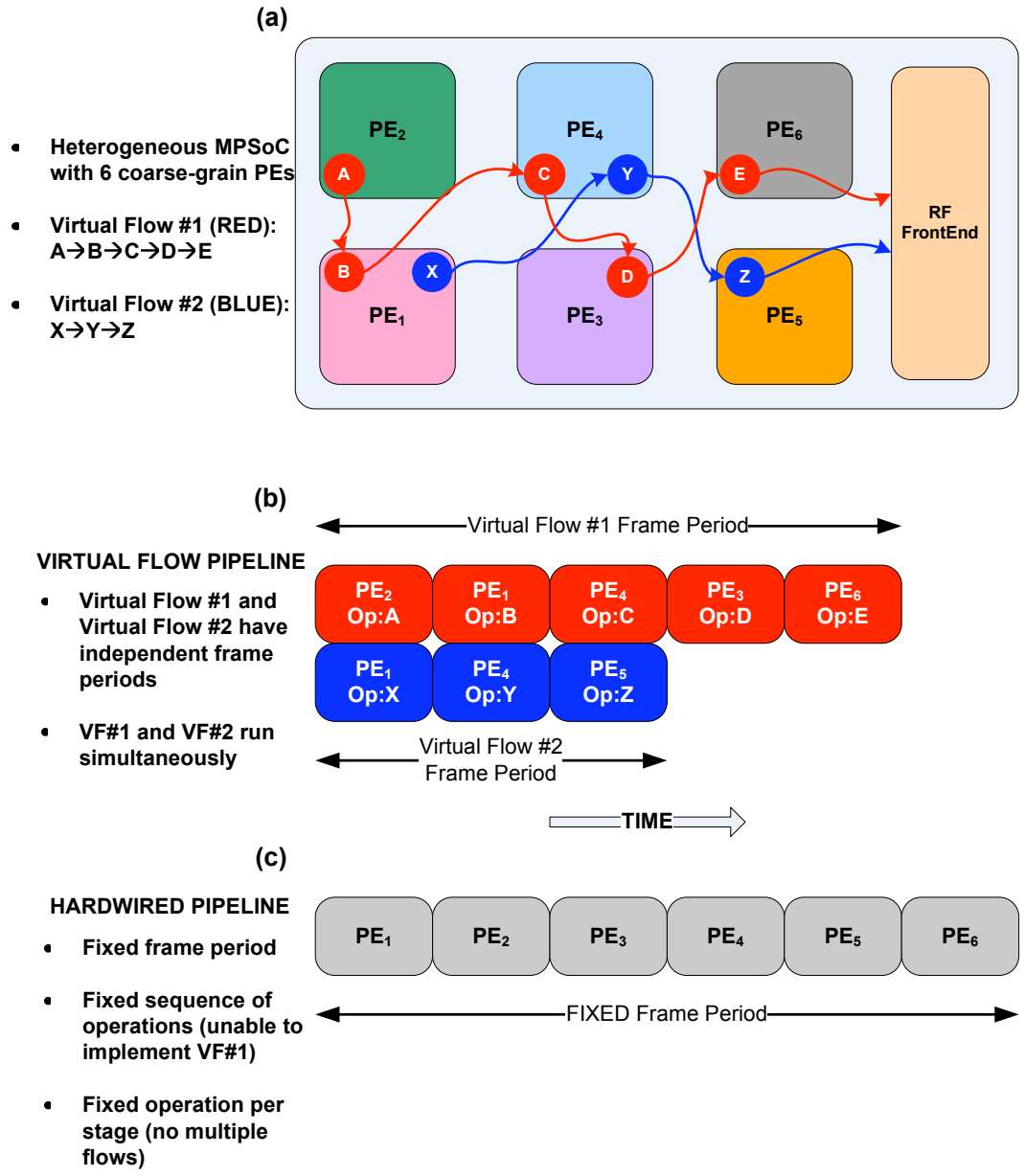


Figure 3.1: Virtual Flow Pipelining – Abstract View

### 3.2.1 Pipelining: Exploitation of Task-level Parallelism

This section considers the execution of the programmed virtual flows in a pipelined manner. Specifically, the VFP framework executes asynchronous pipelines (of *soft nature*) in order to exploit the coarse grained parallelism. Fine-graned instruction and data parallelism is utilized within the individual PEs. As elaborated in section 3.1, the VFP framework uses the divide-and-conquer principle to implement the wireless protocols on a programmable task-level. This is particularly important for 2 reasons;

- it makes the mapping of tasks to PEs and programming the flow simpler (discussed

in section 3.3)

- it presents the opportunity to exploit *task-level parallelism* by executing different operations of potentially independent virtual flows on the PEs in parallel.

In order to exploit this task-level parallelism, the tasks within the virtual flows are executed in a pipelined fashion. For example, the Fig. 3.1b illustrates the pipelined operation of the virtual flows extracted from Fig. 3.1a. The most important advantages to be noted from the Fig. 3.1b are that;

1. The platform is executing two virtual flows concurrently, in a pipelined fashion, by multiplexing the PEs among the flows. Thus, the utilization of PEs is high and high system throughput is sustained, while supporting virtualization.

2. The flexibility of defining the virtual flows enables the latency of the pipelines executing the virtual flows to be variable. This is important from the point of view of achieving the real-time constraints associated with the protocol flows, while maintaining virtualization properties.

### 3.2.2 Virtual Flow Pipelining Overview

The pipelined execution of the programmable virtual flows in essence gives the name *Virtual Flow Pipelining* (VFP). Managing this flexible pipelined operation is achieved using the event-based control-flow sequencing and synchronization mechanisms that constitute the VFP framework. These are discussed in section 4.1.2. The VFP framework also handles the fast context switching and data communication between different PEs (stages of the pipeline) in a manner so as to achieve high PE utilization and system throughput.

In contrast to VFP, a traditional hardware pipeline is incapable of providing the performance, flexibility and virtualization properties. Fig. 3.1c illustrates the inefficiencies if a hardware pipeline was used to implement the two protocol flows depicted in 3.1a. But the benefit of a hardware pipeline is that it has a deterministic performance − since allocation of all the resources is completely inflexible and static throughout the frame

period. The determinism is a desirable feature in order to guarantee QoS features. Our Virtual Flow Pipeline framework adds the required flexibility to the hardware pipelining approach, while retaining performance guaranties. This is achieved in the VFP by use of scheduling mechanisms (discussed in 4.1.2), which respect and enforce the priorities and timings requested by the upper layers.

## 3.3 How are Virtual Flows Created? – The VFP Programming Model

Before we go on and present the VFP control mechanisms, let us consider the VFP programming model. It is the programming model that enables creation and description of the virtual flows – in terms of functionality and performance requirements.

Based on the divide and control principle, our model involves breaking up the protocol flow into constituent tasks/functions. A task is an atomic computational kernel/algorithm that can be completely executed on a PE, e.g. interleaving, MMSE estimation etc. Thus at flow provisioning time, the following information is to be identified and represented in order to create a virtual flow;

1. Description of the tasks involved (i.e. the operation to be executed, number of operands etc.) and their mapping to respective PEs. This is the lower level of programming.

2. Description of control-flow/sequence of the protocol, which effectively stitches the different tasks performed by the PEs into forming a flow. This is the higher level of programming that brings out the data dependencies among tasks.

3. Performance requirements of the flow – in terms of timings and priorities of the constituent tasks from the perspective of higher layers.

The representation of the above information in form of control data structures is called a *Virtual Flow Program*. Multiple such flow programs can be created and defined at the flow provisioning time. But as discussed in the previous chapter (2.2), the need for virtualization, run-time reconfiguration and execution of multiple flows, together with maintenance of real-time constraints adds dynamics to the problem of actual scheduling

and execution of tasks. Hence, the potential sequence space and timings of operations in a virtual flow are described at provisioning time, but the actual sequence and timings are decided at run-time − based on the run-time results.

### 3.3.1 Representing the Virtual Flow Program

The tasks and their sequence is then represented using a *directed acyclic graph* (DAG). A DAG representation enables representation and execution of spawning of one-to-many child tasks as well as facilitates joining many-to-one tasks.

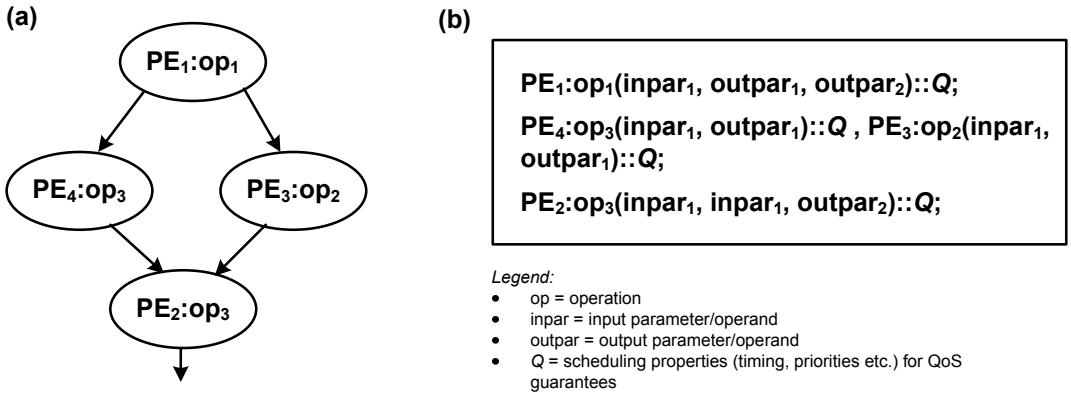


Figure 3.2: VFP Programming Model

For example, lets assume that Fig. 3.2a represents a simple protocol. The structure of the DAG represents the protocol flow (higher level of programmability), whereas the nodes depict the PEs involved and their required configuration (lower level of programmability). Fig. 3.2b, illustrated the example flow program for the protocol depicted in Fig. 3.2a.

On the MPSoC, the virtual flow programs actually translate to control code that resides in the form of tables/data structures in the memories. These tables will be described in the next chapter.

### 3.3.2 Features of the Programming Model

Salient features of the programming model are:

1. The heterogeneous PEs are treated uniformly. Thus, the fact that a PE could a configurable HW module or a programmable processor, does not impact the way the platform is programmed.

2. The programmer is relieved from the burden of synchronization and data dependency checks. This is completely handles by the hardware.

3. Although the mapping of task-to-PE is done by the programmer, the actual scheduling and activation of tasks is managed by the hardware mechanisms at run-time. Thus, the programmer is not required to handle this at compile-time − making the programming easier and more conducive to dynamics associated with virtualization. This feature provides the required level of abstraction towards the higher layers.

4. The task-level break up and two level programmability facilitates pipelining of operations, thus exploiting task level parallelism.

## 3.4 VFP Mechanisms

This section will describe the underlying control mechanisms of the VFP framework. These mechanisms support the programming model and are actually responsible for the execution of the protocol flows on the platform. A unique aspect of the VFP is that these mechanisms are all hardware-oriented. In a way, these mechanisms can be looked at as a hardware implementation of some operating system functions, thus obviating the need for one.

### 3.4.1 Control-flow Sequencing and Data Communication

The control-flow, which executes the higher level programming, is implemented as a sequence of producer-consumer interactions. Basically, the virtual flow program represented by the DAG is nothing but a bunch of producer-consumer sequences. A PE task acts as *producer* for the tasks following it in the DAG. Essentially, for a given virtual flow, every PE in the DAG acts as a producer (except the leaf node tasks) and every

PE acts as a consumer (except the root).

Thus, whenever a PE task (producer) completes, its consumers are identified and further mechanisms are triggered. This process of identifying and engaging of all the consumers of a particular producer task − as per the Control-flow data structure in the memory − in essence executes the control flow.

The data communication, between PEs is also handled by the VFP framework − again using the producer-consumer notion. The framework has provisions for checking race conditions and data overwrites using semaphores. Moreover, data transfers are performed by *Direct Memory Access* (DMA) engines that relieves the PEs of this burden. This effective separation of computation and communication, improves the PE utilization as well as the determinism in allocation of PE bandwidth.

### 3.4.2 Synchronization

The programming model of VFP allows for forking (for parallel execution) and joining of tasks. This calls for the need for task synchronization. For example, in the case of joining, the start of a task has to be synchronized with the completion of multiple producer tasks. In general, the synchronization mechanisms are needed to trigger/activate particular tasks in the flow only after all their associated dependencies have been resolved.

As mentioned in section 3.3.2, the VFP framework handles the task synchronization independently − i.e. without the need for specific programming. This is done by maintaining *counting semaphores* associated with every task in every flow. These semaphores are initialized at the provisioning time (and periodically reinitialized for repeating tasks), and decremented once on every associated producer task completion. Finally, a semaphore reaching zero is the triggering event for that particular task. All these mechanisms are implemented in dedicated hardware logic, together with memory table entries for maintaining the synchronization states.

### 3.4.3 Scheduling

The stringent performance requirement of wireless protocols needs to be supported at the architecture level with mechanisms that will guarantee processing latency, timely response, and provisioned quality of service parameters. Hence effective scheduling mechanisms are incorporated in the VFP framework to satisfy requirements of individual flows as well as to efficiently share the processing resources between the flows.

As mentioned earlier, the virtual flows (tasks, their sequence and performance requirements) are defined at provisioning time, but need the run-time scheduler in order to ensure the performance under dynamic system loading. The run-time scheduler is in charge of ensuring both the deterministic and the statistical (average type) performance guaranties, depending on the flow setup. The scheduling function of the VFP controller multiplexes each PE (configurable hardware unit or programmable processor) either based on a time reservation (for deterministic guarantees) or a best effort scheme (for deterministic guarantees). In order to support synchronous framing type of protocols (e.g., time division multiplexing), the flow scheduling information for the time reservation based scheme also specifies the repetition time.

For providing these degrees of priorities, the tasks whose synchronization criteria is met are queued up as either synchronous tasks or as asynchronous tasks (with multiple priorities) for scheduling. The type of tasks depends on the type of performance demanded as per the virtual flow program.

#### Synchronous Tasks

The synchronous tasks are associated with the deterministic guarantees. These tasks have allocation per repetitive flow frame intervals, reserving the corresponding PE for the required processing time at the specific interval within the periodic frame. In short, synchronous tasks have a dedicated share of the PE bandwidth. Fig. 3.3 illustrates the allocation of bandwidth to the synchronous tasks. These allocations can belong to different virtual flows.

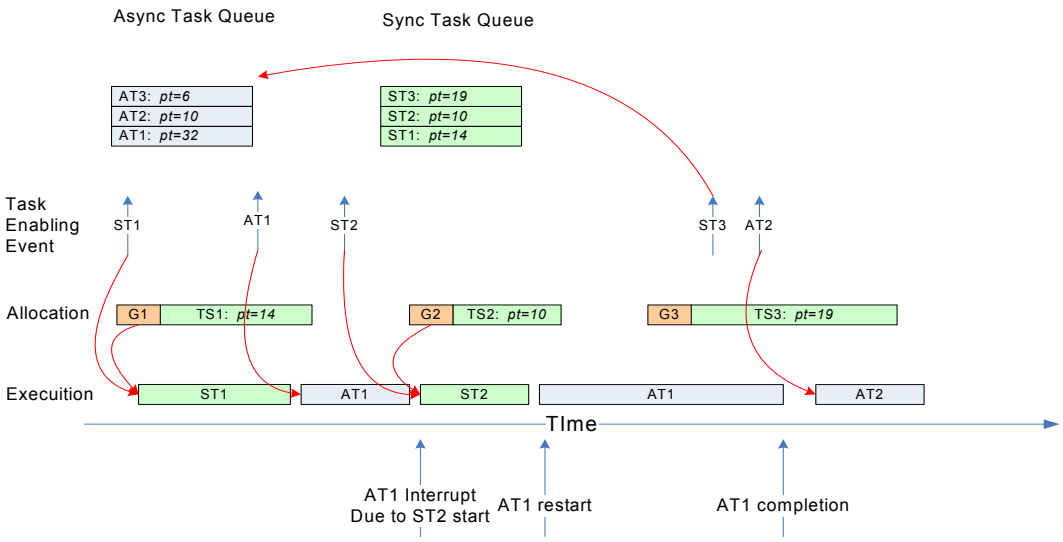


Figure 3.3: VFP Scheduling Policy

### Asynchronous Tasks

These tasks are associated with the statistical guarantees. The asynchronous tasks which are allocated in the shared processing bandwidth left over after the allocation of the reserved capacity by the synchronous tasks. Thus they are scheduled as per a best effort policy. The asynchronous tasks are scheduled by fixed or Weighted Round Robin scheduling discipline. The asynchronous tasks are interrupted at the start time of the synchronous task time slot, and resumed at the point of interruption after the completion of the synchronous task. The interruption is also necessary in order to guaranty the processing bandwidth for the deterministic processing of synchronous tasks. (Presently our implementation does not support interruption of async tasks. Instead we stall scheduling an async task if its processing time is long enough so as to clash with a synchronous task allocation.)

The fig. 3.3 illustrates all the scheduling policies. Important observations are that the async task AT1 is interrupted and resumed later in order to cater to the dedicated bandwidth of synchronous task ST2. The G1, G2 and G3 are guard times for the

synchronous tasks – it is the time window for sampling the enabling event. Enabling events delayed more than the time G cause the synchronous task to be rescheduled (red arrow) and the dedicated bandwidth is released (used up by AT2).

### 3.4.4 Context Switching

Given the short processing time for the packet, or parts of it, the fast context switching needs to be supported by both software execution (CPUs), as well as hardware PEs, otherwise the utilization of the units will be low. For the systems targeting hundreds of megabits per second data throughput the unit processing time is sub-microsecond, dictating the use of hardware assisted and controlled context switching.

Also, the processing time of the task execution has to be deterministic. In order to cater to these requirements, the VFP system controller completely handles all the activities up to beginning of task processing (pre-processing) and takes over immediately after the completion of task execution. Thus the PEs are engaged only in useful processing activities and can be kept occupied in a pipelined fashion.

The pre-processing functions performed by the VFP for the PE, include scheduling the task, bringing in the input data, setting up the input buffer pointers and flow context information. Following this the PE is triggered with the task command, which it promptly begins executing. Similarly, the post-processing functions performed by the VFP after the task execution completes include, identifying consumers and transferring the output data.

Thus, the constituent PEs of the system have a very myopic view. They are unaware of notions of flows nor do they know to which flow the task they are processing belongs. The PEs just respond to the commands from the VFP controller and hand over the control back to the VFP controller as soon as execution completes.

## 3.5 Putting it All Together – The Layered Radio Perspective

This sections attempts to provide a high level overview of the VFP framework while summarizing the features and mechanisms discussed in the previous sections. We try

to provide this complete picture by proposing a *Layered Radio Architecture* view of the system. This view is depicted in the fig. 3.4.

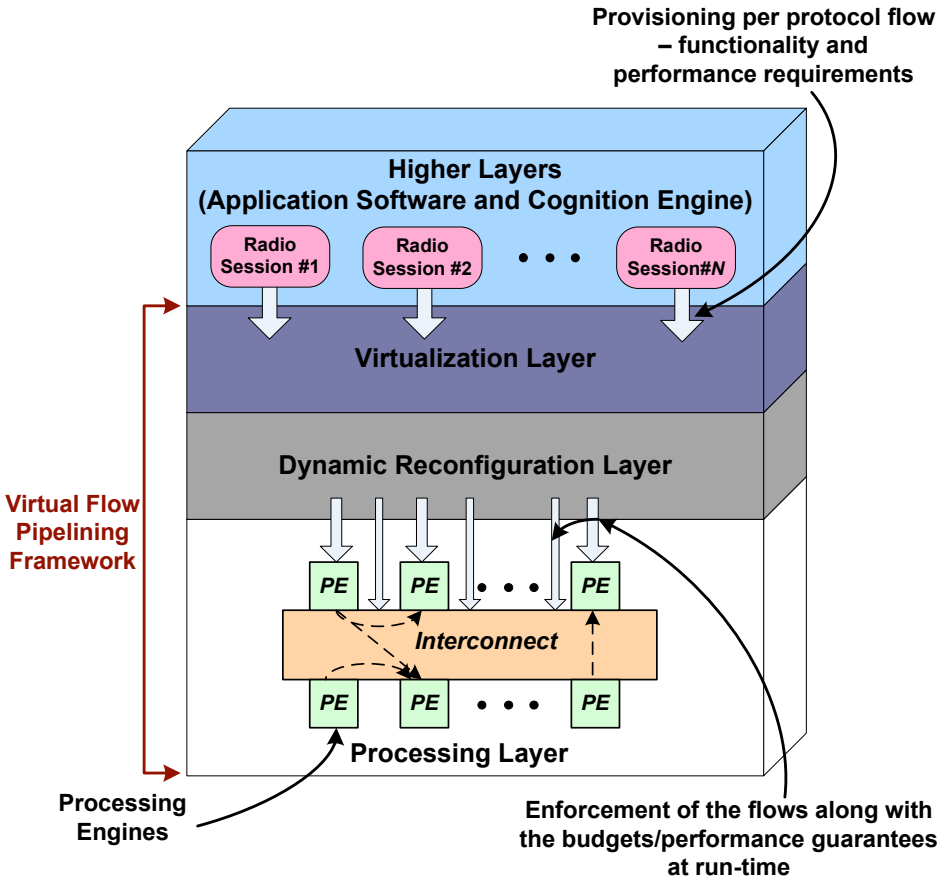


Figure 3.4: Layered Radio Architecture

The following points describe the different layers while relating them to the VFP:

**Higher Layers** These layers dictate the number of flows executed by the lower layers as well as their functionality and performance requirements. This is done by creating the virtual flow programs (according to the protocol requirements) through interactions with the *virtualization layer*.

**Virtualization Layer** This layer presents the uniform API for the higher layers to program the virtual flows. It effectively provides a hardware abstraction between

the multi-radio environment at the higher layers and the common hardware platform at the lower levels − thus shielding the radio applications from the sharing and loading of the processing layer and hiding the existence of other concurrent seesions.

**Dynamic Reconfiguration Layer** This layer constitutes the efficient control-flow, synchronization, fast context switching and run-time scheduling mechanisms as discussed in section 4.1.2. It is this layer that is responsive to the dynamics of the system and is responsible for enforcing the budgets and guarantees provisioned at compile time. On one side the dynamic reconfiguration layer interacts with the virtualization layer to receive the provisioned functionality and timings, while on the other side it schedules and drives the PE execution and data transfers as per the run-time results and events. It is also responsible for exploiting task-level parallelism by managing the asynchronous pipeline.

**Processing Layer** This is the layer where the actual data plane processing is done. It constitutes the heterogeneous PEs (which presents a uniform interface to the dynamic reconfiguration layer) and the underlying interconnect. This layer offers the dynamic reconfig layer with three degrees of freedom in order to achieve the required flexibility. They are;

1. selection of PE configuration, which decides what operation/algorithm the PE executes;
2. sequencing of the PEs to form the coarse grain protocol flow; and
3. denition of new PE functions by adding new software programs to the CPU-based data plane PEs.

The constituents of the processing layer have a myopic view. They are completely transparent of the notion of flows. They are like slaves of the dynamic reconfiguration layer − responding to commands on as a case by case basis.

## 3.6 Correlating the VFP Framework and the Requirements

In this final section of the chapter describing the VFP framework, we will correlate the framework with the requirements covered in Chapter2. This will bring out the significance of different aspects of the framework and help evaluate it qualitatively. Let us discuss each of the requirements, one by one.

**Platform-based Paradigm** This requirement is satisfied by the two level programming model provided by VFP. It provides separation between programming the PE tasks and the actual top level sequence of the tasks − effectively achieving separation of computation and communication. Uniform support for heterogeneous components further qualifies the framework.

**Programmability/Configurability and Dynamic Reconfiguration** The Programmability requirement is satisfied in a straightforward manner by the programming provisions. Various protocols/standards can be programmed onto the SoC at compile time. The particular protocol to be used/executed in order to transmit a particular packet can be decided at run-time − when inserting the packet for baseband processing. This feature in effect achieves the required per-packet dynamic reconfigurability.

**Multiple Simultaneous Traffic Flows** The support for having multiple diverse protocols pre-programmed together with the ability of the framework to multiplex and interleave the tasks onto PEs achieves this requirement.

**Virtualization and QoS guarantees** This is achieved by the scheduler policy and its run-time implementation. The protocols are pre-programmed and then dynamically initiated by the higher layers at run-time − without system loading considerations. Then the run-time scheduling of the tasks to PEs, which execute them with a myopic view, achieves the virtualization.

**Throughput, Real-time Constraints and Power Efficiency** These are achieved by all the VFP mechanisms (scheduling, synchronization, context s/w) and their

unique hardware implementation. These mechanisms offload the PE of all the task and flow management functions. Thus, the PEs are kept busy with only useful application/algorithm processing. Furthermore, the pipelining (task-level parallelism) further increases throughput. This hardware-based task management together with the scheduler policy and transparent data passing, increase the determinism in the system − a key requirement for meeting real time constraints. Avoiding the use of operating system or a CPU operating at very high speed helps power reduction.

# Chapter 4

# Virtual Flow Pipelining based SoC Architecture

In Chapter 3 we have presented the *Virtual Flow Pipelining* framework for satisfying the requirements of programmable and flexible radio processing. In this chapter we will eloborate on the development of the framework into a *system-on-chip* (SoC) architecture. Putting it differently, the previous chapter answers the "What" questions (What are the specs, the mechanisms, the programs?). This chapter will answers the "How" questions (How is the framework implemented? How is it organized?).

## 4.1 Preliminaries

In this section we put forth the preliminaries that lead to the SoC architectural approaches and micro-architecure discussions. Strictly speaking this section while beginning to answer the *Hows* actually extends on the *Whats* discussed in the previous chapter. The reason for including these here is that they are closely associated with and naturally lead to the implementation discussions.

### 4.1.1 Control-code Memory Structures

We begin with showing how the virtual flow program code is actually mapped onto the memory. In a way this answers the − What are the control code memory data structures? − question. As mentioned in section 3.3, the protocol flows are represented as directed acyclic graphs. Two such protocol flows are shown in Fig. 4.1. The tables/data structures representing these flows in the memory are also shown in Fig. 4.1. We use two flows to demonstrate how multiple flows reside in the control memory. As can be seen the figure shows 2 types of structures; Task Descriptors tables and table for Scheduling Properties and Synchronization State.

While the Task Descriptors deal with the DAG description and info pertaining to the execution of the respective task, the other structure holds information required for the consistent and correct operation of the flows together with the maintenance of the required timings and QoS across flows.

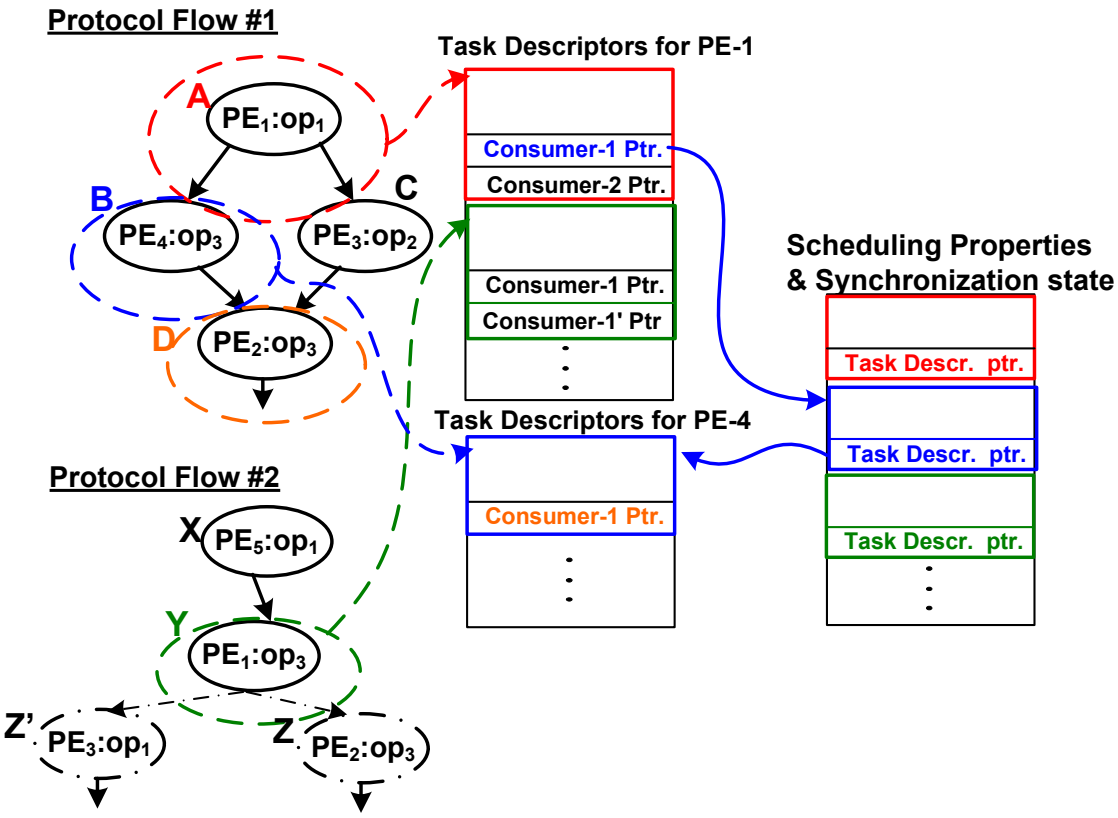


Figure 4.1: Control Code Data Structures

### Task Descriptors

It the *Task Descriptors* (TD) that describe the DAG. There exists a TD entry for every task in every flow. The information in each TD falls in two main categories – first, context switch and task execution related information and second, control-flow related information. We elaborate on each of these categories next.

**Context Switch and Task Execution Related Info** This part of the task descriptors contains all the information needed by the PE before it can start the task

execution. This can be thought to be the information at the *nodes* of the protocol flow DAGs. Explicitly, this information is;

1. Task Command and type: Tells the PE what command is to be executed. Other flags also convey what type of task it is, i.e., chunking task, first chunk or last chink etc. (Chunking will be explained in the later sections).
2. Input Data Info: Provides information about how many buffers in the input memory are associated with this task. It mentions the pointers and size for each of the buffers where the actual data to be operated on resides.
3. Flow Context Info: Is used to appropriately resume tasks that were interrupted or tasks which are operated on in parts (e.g. chunks). This information is just the pointer and size of the location in the internal memory of the PE where the actual flow context info resides.

**Control-flow Related Info** This part of the task descriptors provides the information about the consumers of the task being described. In effect, this can be thought of as the information pertaining to the *edges* of the DAGs. It contains the IDs of the consumer tasks in the flow together with the pointers to the locations of the respective input memories where the output of the current task should be copied. This control-flow information in effect describes any forking and/or joining of tasks. It also facilitates having potential consumers, where the actual run-time consumer will be decided based on the result of the current task. This feature, in effect, enables programming simple *if-else* type scenarios. As shown in the Fig. 4.1, for Flow−1, task A has two consumers (forking) − B and C. The TD for task A has pointers for each B and C. The Flow−2 illustrates the case of potential consumers, task Y can have one consumer − either Z or Z' − depending on the result of task Y.

Each TD entry has a link to its corresponding entry in the Scheduling Properties and Sync. State tables − not shown explicitly in Fig. 4.1, its just color coded. In the Fig. 4.1, the TDs are shown as separated per PE. They can be such in physical

separate memories per PE or just in one memory. For the Distributed-control approach discussed in section 4.2 the TDs are in physically separate memories per PE.

**Scheduling Properties and Sync. State tables**

These tables too have an entry per task per flow. As the name suggests, each task entry contains two categories of information.

**Scheduling Properties** This part holds all the info needed by the scheduler for appropriately scheduling the task for meeting the performance guarantees and timings. This info here is used while queuing the task in the queues of scheduler − which then makes the scheduling decisions as per the policy. This info indicates whether the scheduler class of the task − Synchronous or Asynchronous (section 3.4.3). For Synchronous tasks it includes task start time, its reschedule period and repetition count, type of task (e.g. chunking, first chunk or not etc.), chunk size, TD pointer, guard time window width etc. For Asynchronous tasks, it provides the priority of the task, whether it is control or data tasks, its processing time and TD pointer.

**Synchronization State** It holds the semaphores and flags needed for correctly synchronizing the control-flow and maintaining data consistency. It has a counting semaphore for task sync. − especially for managing the joining of tasks. It also has flags for maintaining input data consistency. Copying data to a input buffer is allowed only if the corresponding buffer flag is reset indicating the buffer is available. This prevents overwrites of data, thus maintaining its consistency through the flow. Other info includes initial value of the counting semaphore, used to reload the task.

We will discuss more about the physical partitioning of these control-code memory structures in later sections when we present the architectural approaches.

### 4.1.2 VFP Control Mechanisms

As mentioned in section 4.1.2, the protocol flow is implemented as a sequence of producer-consumer interactions. It is the VFP mechanisms that comprise these producer-consumer interactions. The main objective of the VFP mechanisms is to offload the PEs of all these producer-consumer interactions, thus maximizing the PE utilization for useful application/algorithm processing. In this section we present a step-by-step account of the set of interactions that happen as part of the VFP scheme. So this section answers the question − What are the interactions between producer and consumer? In the later sections, when we discuss the architecture approaches, we will see *how* each of these preliminary steps are implemented and the impact of the chosen approach.

Fig. 4.2 is an abstract diagram that depicts the steps (numbered) between a producer and a consumer. As shown in the figure, a PE together with its local data memory and any more local control logic (e.g. DMA engines) is called a *Functional Unit* (FU). Other elements of the Fig. 4.2 include the control-code memory structures and task scheduler queues. The scheduler queues and task descriptors are shown as separate and at the depicted positions just to convey the idea that they are associated per PE. Their positions and/or separation in Fig. 4.2 does not imply anything about their actual physical position and organization. This figure is an abstract representation only for the purpose of illustrating the interactions.

Following points elaborate on the steps;

We begin with the assumption that the Producer FU already has a few tasks queued up in its scheduler queues, with their corresponding input data residing in the local data memory.

1. As per the scheduling policy a task from the ones present in the producer scheduler queues is scheduled for execution whenever the producer PE frees up, i.e., completes its previous scheduled task.

2. Before the scheduled task can be actually triggered/dispatched for execution to the producer PE, we need to context switch. As explained in section 4.1.1, all the information required to perform the context switch and trigger the task is present

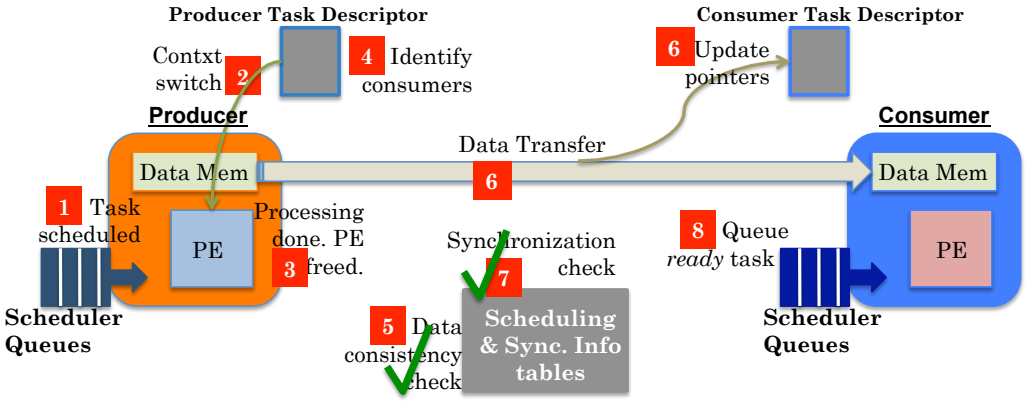


Figure 4.2: VFP Control Mechanisms

in the Task Descriptor for the respective task. This includes passing the pointers and sizes of the input data, providing the actual command to be executed, and passing pointers to any residue flow context information. Once this context switch setup is completed, the PE is triggered to begin processing.

3. As soon as the processing is completed the PE is freed up. The PE is immediately ready to take up a new task selected/scheduled from the candidates in the scheduler queues of associated with the producer. Thus, a new task could be processed in parallel with all the steps to follow – maximizing the PE utilization. The synchronization state in the Scheduling Properties and Sync. State table entry of this completed task is also reset/reloaded – i.e. the task sync. counting semaphore is reloaded and the data consistency check flag is cleared to indicate its availability.

4. Next the consumers of the task just completed are identified, using the control-flow information residing in the task descriptor. The actual consumers are chosen from a set of potential consumers (equivalent to resolving if-else dependency) depending on the result of the PE. For example, for the Decoder-PE, the consumer may depend on whether the CRC passes or fails. These kind of control dependencies

are resolved using a vector-based scheme discussed and detailed in [31].

5. Having identified the consumer, an input buffer availability check is performed for each consumer. This check helps maintain data consistency by preventing overwrites. The flags for performing this check reside in the entry corresponding to the consumer, in the Scheduling Properties and Sync. State tables.

6. After the data consistency check for a consumer passes, output data from the producer FU is copied to the consumer input data memory. This transfers happens using DMA engines − independent of the producer or consumer PEs. The consumer task descriptor is updated with the pointer and size of the data just received.

7. Following the data transfer, the task synchronization is performed. This involves updating (decrementing) the counting semaphore in the consumer task's Scheduling Properties and Sync. State table entry. If the semaphore has reached zero means all the producer tasks, joining at this consumer task, have completed. Such a consumer task is said to be *ready* for execution.

8. If the task synchronization condition in the previous step is satisfied, then the *ready* consumer task is queued up in its respective scheduler queue using the scheduling properties in its corresponding entry.

9. This completes the set of interactions between a producer and a consumer. This consumer assumes the role of producer for the tasks that follow it. The same set of steps, starting from step number 1 above, can be imagined to be performed by it as well.

Thus, the producer-consumer interactions are performed by dedicated hardware in parallel with the actual processing of PEs. This in effect achieves a pipelined operation of the PEs, which exploits the task-level parallelism offered by wireless protocols.

## 4.2 Distributed-Control Approach

Having discussed the preliminaries, this section will present an approach that was first considered and implemented for the VFP framework. We will first present the SoC organization and then discuss the issues that this approach had.

### 4.2.1 Distributed-control SoC Organization

Fig. 4.3 depicts the SoC organization for the approach using distributed control.

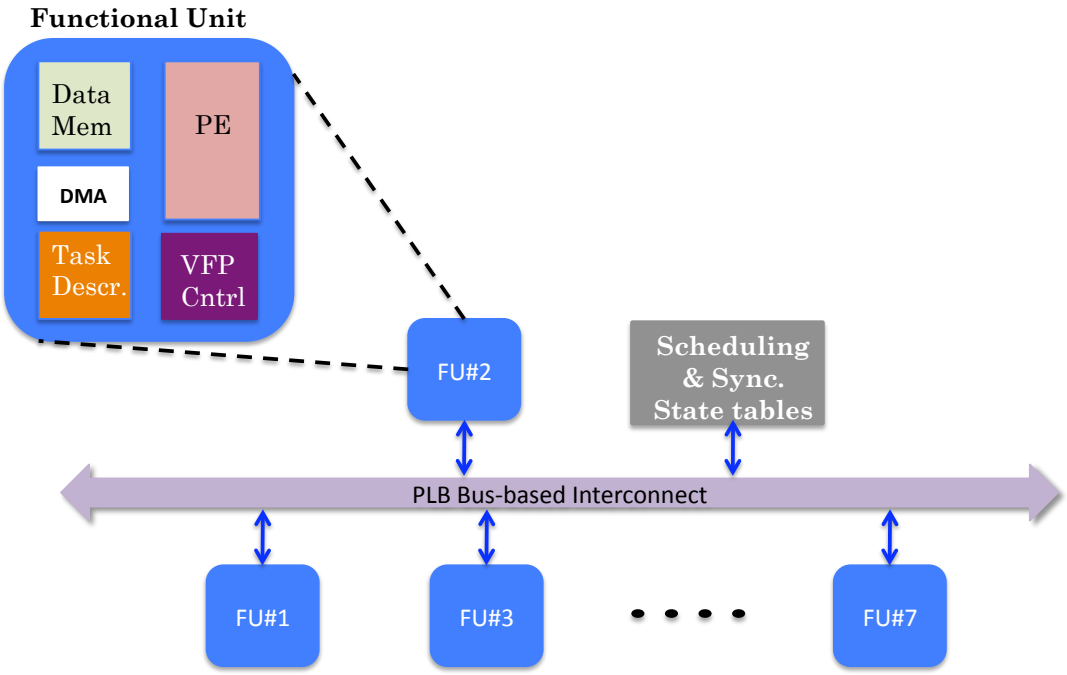


Figure 4.3: Distributed-Control SoC Organization

Following are the points to note about the organization;

- The PEs have associated local data memories and data is explicitly passed between the data memories, without the use of any central shared data memory. This organization avoids the need for caches, which may be needed if a large and slow

shared memory is used. Avoiding caches simplifies the memory management and also makes the PE processing more deterministic − because data is available locally.

- The VFP control is completely distributed, with a dedicated VFP control module associated with every PE. This control module also includes a DMA engine for performing the producer-to-consumer data transfers.

- The Task Descriptors associated with each PE are physically separate memories. These TD memories are local to the corresponding VFP control modules. Although the TD memory is local to the VFP module it also has a port connected to the global SoC interconnect. This serves 2 purposes − first, for populating the entries at flow provisioning time, and second, for enabling remote VFP modules to update values in it at run-time.

- The Scheduling Properties and Sync. State tables for all the tasks of all the PEs reside in one physical shared memory. This memory is connected to the global SoC interconnect so that it can be accessed (updated and read) by all the distributed VFP modules − thus permitting any PEs to form producer-consumer relations.

- Thus, the overall organization consists of a number of *functional units* (FUs) connected using a PLB bus based interconnect. The FU comprises of the PE, the data memory (input and output), the VFP control module and the task descriptor memory.

Next let us discuss the issues with the distributed architecture that have necessitated the need for a new architecture.

### 4.2.2 Issues with the Distributed Approach

The distributed approach has the following drawbacks that impact the system scalability and performance.

### High Hardware Complexity

The hardware complexity of the VFP controller is very high − approximately 25K gates. This number is larger that some of the simple PEs, resulting in more than 100 percent overhead. Moreover, the distributed architecture has a VFP controller per PE, which limits the scalability of the system. This is because the total hardware overhead becomes prohibitively large as we increase the number of PEs in the system − its a linear increase. This is not good, especially while considering solutions for infrastructure equipment devices which have to accommodate large number of PEs for supporting hundreds of traffic flows simultaneously.

## Control Communication Bottleneck

The producer-consumer interactions for the distributed architecture are producer-diven. This means that once the producer task completes the producer VFP controller is responsible for performing all the control mechanisms, right up to queuing of the *ready* task into the consumer side scheduler queues. But because the VFP controllers are distributed, the producer driven mechanisms result in considerable amount of control traffic on the global interconnect. This traffic comprises of the following accesses (Fig. 4.4):

1. Accesses to the shared Scheduling Properties and Sync. State tables memory to read the data consistency check semaphore of the consumer as well as update its own. Also to update and read the task synchronization semaphores after the output data has been passed to the consumer. If the synchronization enables the consumer task, then its scheduling descriptor is fetched from the shared tables too − all over the global interconnect.

2. The producer VFP controller also needs to update the task descriptors at the consumer side with the pointers and size of the data passed to it − again over the global interconnect.

3. Finally, after the task synchronization passes, the producer VFP is responsible

for queuing the *ready* task into the consumer scheduler queues. This also needs the global bus access.

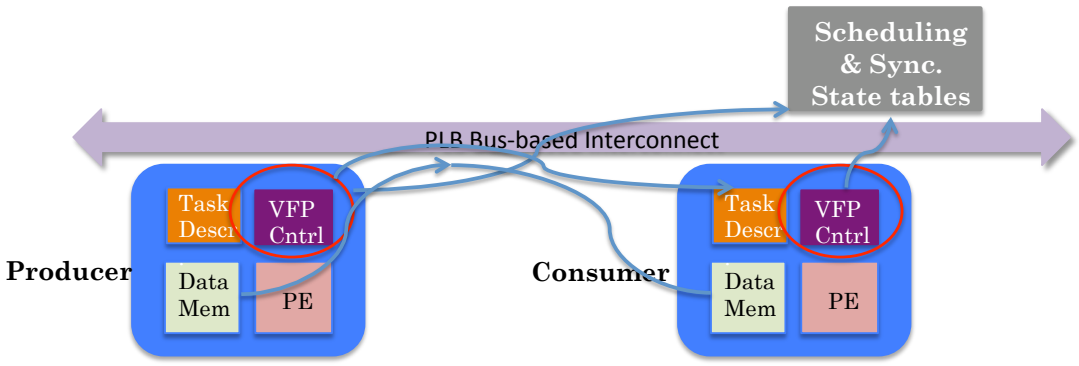


Figure 4.4: Issues with the Distributed-Control Organization

Thus it can be seen that each producer-consumer interaction results in considerable global accesses. Moreover, this kind of global control communication traffic is generated by multiple PEs in the system. The accumulative effect is that the control communication is delayed and it also impacts the interconnect bandwidth available for the data transfers. Even if the there is sufficient bandwidth for the traffic, these global accesses are high latency accesses, i.e., approximately 10-15 cycles of overhead (DMA engine + bus arbitration). This means short accesses such as fetching the data consistency semaphore (which could be a polling operation), has a 10 times high overhead associated with it.

Thus, with its high hardware overhead, the distributed approach is not an efficiently scalable solution. The communication bottleneck not only adds to this but also impacts

the system performance by increasing latency and effectively reducing PE utilization for short processing tasks.

## 4.3 Clustering-based SoC Organization

The issues with the fully distributed approach call for a new SoC organization. But the diametrically opposite solution of having a fully-centralized organization has obvious limitations of scalability. The need for striking a balance between the scalability and hardware overhead has resulted in the new clustering-based organization shown in Fig.

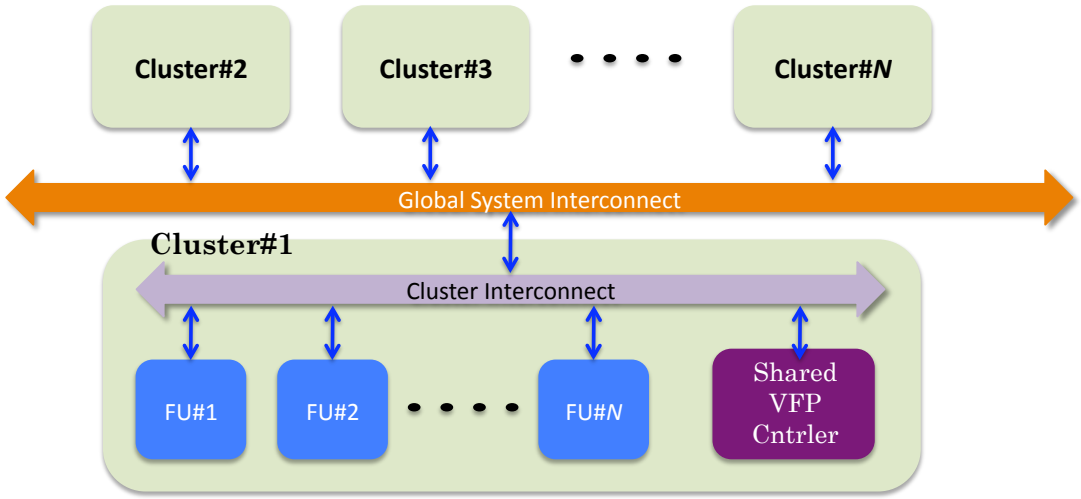


Figure 4.5: Clustering-based SoC Organization

There are 3 main challenges that associated with the clustering-based architecture. They are;

1. Designing a centralized VFP controller that will be shared across multiple PEs in the cluster.

2. Providing a solution for efficient control communication between the PEs and the

VFP controller.

3. Maintaining scalability by handling the possibility of having producer-consumer interactions across cluster boundaries.

In the following sections we discuss how each of these challenges is handled.

## 4.4 Shared VFP Controller

As discussed in the previous section a single VFP controller will be used to serve all the PEs in a cluster. This will help reduce the hardware complexity of the system. The challenge in designing such a shared VFP controller is that the performance impact of sharing must be minimized. Otherwise, such solution will not be very different from a Software Controlled approach discussed in Chapter 1.

### 4.4.1 Control-code Memory Organization

Preliminary modification with sharing the VFP controller is to localize the control data structures in the system. Since most of the control mechanisms are now centralized, the associated control-code memories can be made local as well. Thus the accesses to these memories can be local and hence have very low latencies − 1-2 cycles. This also reduces the control communication across PEs, which was a problem with the distributed SoC. Specifically, the memory organization changes are;

- The Scheduling Properties and Synchronization State tables are now provided with a local port to the VFP controller. These tables contain the entries for all the tasks of every PE that belongs to the cluster in which the VFP controller is shared. As we will see in the next section the table can be accesses by multiple blocks in the VFP simultaneously, hence we need a simple fair round-robin arbiter for controlled access to the memory.

- The Scheduler Task Queues that were previously a distributed are now centralized in the VFP controller.

- The control-flow information is extracted from the Task Descriptors and is made local to the shared controller in the form of new control structure called Task-flow tables. Also, the link to the control-flow table entry is added to the Task Descriptor of the corresponding task.

- Finally, the Task Descriptors which now contain only the task operation information, e.g., command, data pointers and size etc. are still kept distributed along with the associated control logic hardware. This is done in order to have fast dispatch of the task once it has been scheduled.

### 4.4.2 Parallelism among the VFP functions

As discussed in section 4.1.2, the main control functions of a VFP controller are;

- Task scheduling
- Context switch
- Consumer identification
- Data consistency check
- DMA transfer
- Task synchronization

These constituent control mechanisms of the VFP framework present a scope for exploiting parallelism. This parallelism is illustrated in the Fig. 4.6. The first row in the figure shows the sequence of the VFP control functions associated with PE-1. The fields with an offset, i.e., *Context switch and Task processing* and *PE output data transfer*, are the functions that are not performed by the shared part of the VFP controller. They are instead distributed. The second and third rows in the figure depicts the sequence of VFP functions for PE-2 and PE-3. The Fig. 4.6 can be thought of as a pipelining sequence diagram. It can be seen that at a given time instant, each of the three PEs are performing/executing different VFP mechanisms. Thus by partitioning the design of the shared VFP controller into these potentially concurrent VFP functions, this parallelism

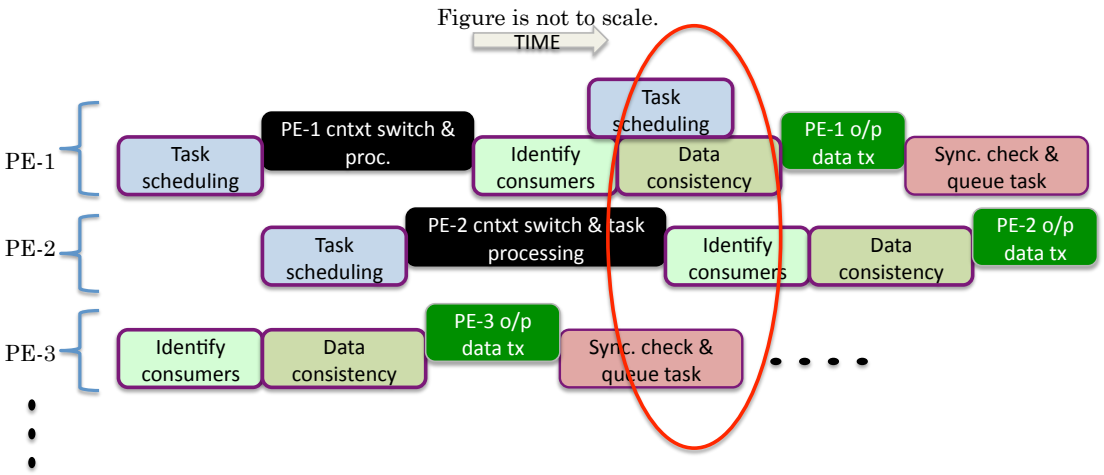


Figure 4.6: Parallelism Among VFP Mechanisms

can be exploited. Such a design can then serve multiple PEs simultaneously, reducing the performance impact of having a shared controller.

### 4.4.3 VFP Controller Architecture

As discussed in the previous section, exploiting the function-level parallelism is the main strategy for improving performance of the shared VFP controller. The Fig. 4.7 shows an abstract view of the microarchitecture of the VFP controller. The microarchitecture is composed of macro-pipelined stages. The structure is an asynchronous buffered pipeline. The pipeline stages have short elastic buffers – FIFOs – between them. These buffers are indicated as brown rectangles with arrowheads in the Fig. 4.7. The main stages in the pipeline are – Consumer identification (triggered by the PE after a task completes), Data consistency check, DMA transfer, Task synchronization and *ready* task queuing, Task scheduling, and finally Context switch after which the Task processing begins. The data transfer is performed in a distributed fashion. The VFP controller delegates the data transfer task to the distributed DMA engines – one per PE – which on completion of the transfer get back into the VFP controller pipeline using messaging.

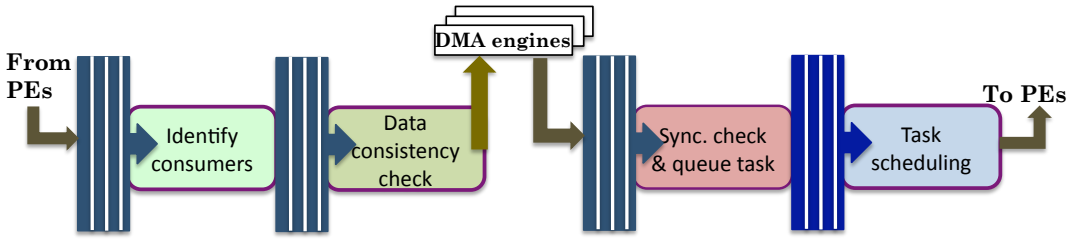


Figure 4.7: Shared VFP Controller Pipelined Architecture

Each of the stages serve different PEs simultaneously. As seen in the Fig. 4.8, at some time instant #1, various PEs are engaging separate stages of the controller. Also, PE-b and PE-c are performing data transfers using the independent DMA engines. At a later time instant (called #2), seen is Fig. 4.8, different PEs are engaging the stages. Thus at each time instant, multiple PEs are served. Each of these stages exploits more parallelism internally, but is not depicted in the figure for clarity.

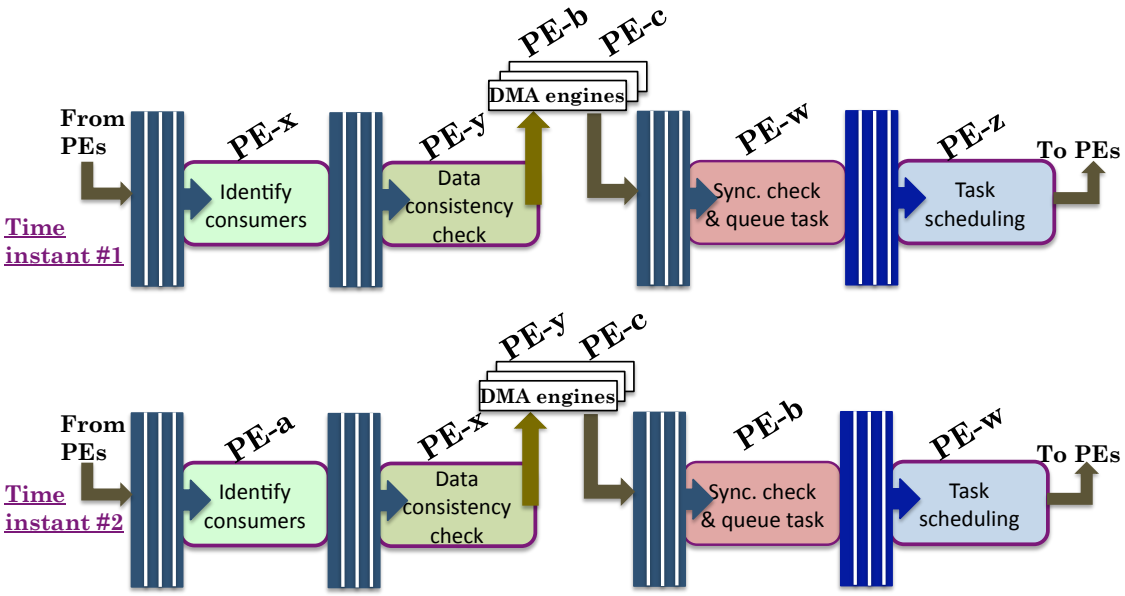


Figure 4.8: Parallelism Exploited by the VFP Controller Pipeline

Thus it can be seen that the shared VFP controller has *explicit hardware level parallelism*. This is the major advantage over the Software Controlled approach –

which can at most have pseudo-parallelism of multithreading. Moreover, each of the stages in the VFP controller is designed specifically to perform the associated control functions. This optimization is a further performance advantage.

The detailed design with internal blocks, state machines etc. can be found in [32, 33].

## 4.5 Control Communication Interconnect

The shared VFP controller requires to communicate with all the PEs within the cluster and vice versa. Moreover, since the VFP is pipelined and simultaneously serves multiple PEs, there arises a need to keep track of the PE being served. This is done by introducing the concept of IDs for each of the PEs. In fact, the IDs are hierarchical − 8 bit values − higher 4 bits specify the cluster number and the lower 4 bits specify the FU ID within the cluster. It is using these IDs that the VFP associates the messages moving along the pipeline stages with the appropriate PEs. IDs also enable the VFP to communicate with the appropriate PE.

Another issue with the control communication between the PEs and the VFP controller is that if it happens over the cluster interconnect, then we essentially have issues similar to the distributed approach. Moreover, since the controller is now a remote entity with respect to the PE, the control communication has to messaging based. This means that each PE needs to create a short 2-4 word message containing its ID and all other information and pointers required by the VFP controller to perform the delegated function. The detailed message formats for different types of messaging between the VFP controller and the PEs is present in [34, 35].

If this messaging together with the data communication happens over the cluster interconnect, it will be a potential bottleneck. In fact, even if we put aside the interconnect bandwidth issue, the latency for transferring each 2-4 word message will be approximately 16 plus clock cycles (through the DMA, arbitration on either side etc.). Hence, the need for a new fast control communication interconnect, which will also offload the cluster interconnect dedicating it for data transfers only.

Considering this requirement five simple and fast customized buses are designed

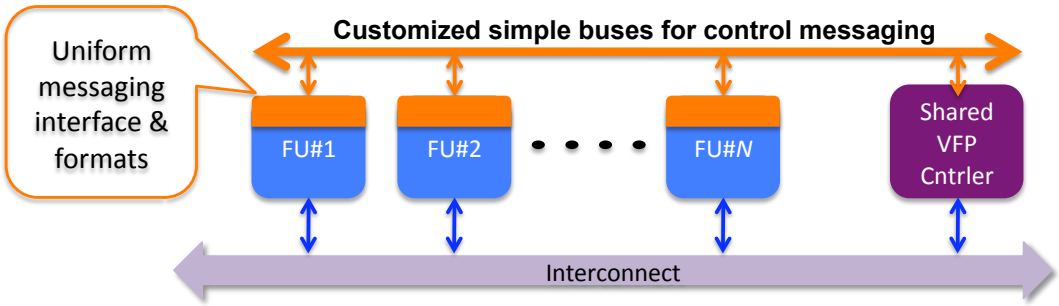


Figure 4.9: VFP-FU Control Communication Interconnect

connecting the PEs and the VFP controller – depicted in Fig. 4.9. The buses are simple and low overhead because they are just one-to-many or many-to-one type of buses. Hence, at most a decoder (using the FU ID) or arbiter is required per bus to select among the PEs in a cluster to communicate with. The arbiters designed are simple fair round-robin arbiters so that each PE is served equally by the VFP controller. These buses are fast because they have a dedicated interface at each PE and VFP controller. Hence, there is no arbitration within the PE or VFP controller – the only arbitration is between PEs – which takes just 1-2 cycles extra. Moreover these buses are customized in terms of their width and communication protocol – resulting in ready to use information in the messages. All these features, reduce the latency involved in the PE-to-VFP communications.

This control communication interconnect requires uniform interface to be designed per FU. Also, the message formats have to be uniform across FUs. The FU interface specifications and design along with the message formats are detailed in [34, 35]

## 4.6 Clustering-based Producer-Consumer Interactions

As elaborated in the section 4.1.2, the VFP mechanisms essentially comprise of producer-consumer interactions. For the case of the distributed-control SoC organization, as discussed in section 4.2.2, these control mechanisms are driven by the producer side VFP. But with the adoption of a clustering based organization we now have two possible scenarios;

**Intra-cluster producer-consumer relationship:** Scenario when the producer and consumer are both in the same cluster.

**Inter-cluster producer-consumer relationship:** When the producer and consumer are in separate clusters.

### 4.6.1 Intra-Cluster Producer-Consumer Interaction

Since both the producer and consumer are in the same cluster, it is fairly obvious that the VFP controller of the cluster is responsible for handling the interaction mechanisms. Also, the control data structures associated with the producer as well as the consumer tasks are all present locally to the VFP controller of the cluster. The producer-consumer interactions in this case are carried out as per the pipeline flow as shown in the Fig. 4.7. Basically, once the producer tasks completes the consumer has to be identified. The Task-flow tables data structure, which is now local to the VFP is used for this purpose. But, it contains the potential consumers. The actual consumers depend on the result of the producer task. Here, the PE generates a consumer identification vector which depends on the result of the computation. The vector is then deciphered by the VFP consumer identification stage to select the actual consumer from the potential list in the Task-flow tables. After the actual consumer has been identified, its ID is available. The VFP controller consumer identification stage uses the upper 4 bits (cluster ID) in the ID to decide whether the consumer belongs to the same cluster or not. Once, it is identified that the consumer belongs to the same cluster, the control is just passed on to the next stage, i.e., Data consistency check, within the same VFP controller. This is done by writing a message/descriptor into its queue. The rest of the control

functions continue as per the sequence of pipeline stages discussed in section 4.4.3. The data transfer is delegated to the distributed DMA engine (of the consumer PE) within the same cluster. The data is then read by the consumer side DMA into its inpu memory from the producer side output memory. The reason for this (consumer-driven) manner of data transfer is explained in the next section. The data transfer happens across the cluster interconnect. The complete VFP control latency from the point a producer completes its task to the point the consumer task initiates is approximately 120 clock cycles. This number is considering the DMA engine latency as well, but ignoring the time taken to transfer data from producer to consumer. This data transfer time adds to the latency. From the point of view of throughput, the VFP mechanisms aim at parallelizing the processing of a task with the data transfer of the previous task, thus improving throughput. For this we use two output buffers (ping and pong) in an alternating fashion − more about this will be discussed in the Chapter 5

### 4.6.2 Inter-Cluster Producer-Consumer Interaction

In this scenario the producer and consumer are in separate clusters. Hence, the question arises − which VFP controller should take responsibility of performing the control functions?

The function of Consumer identification has to be performed by the producer-side VFP. In fact, it is only after identifying the consumer and evaluating its cluster ID is it realized that this is a inter-cluster producer-consumer scenario.

Once the scenario has been identified, the control is transfered to the consumer-side VFP. How this is done is discussed later, but first lets discuss the reasons for this decision. The memories storing the control data structures for a particular consumer PE are present in the same cluster and local to the cluster's shared VFP controller. This is because most producer-consumer interactions are confined to the same cluster. Then the major reason for having a primarily consumer-driven VFP control is that it maintains all the control related memory accesses to local interactions − thus maintaining the benefits of having control memories local to the VFP controller. In contrast, if we had a producer-side VFP driven scheme, it would need global accesses by the VFP controller

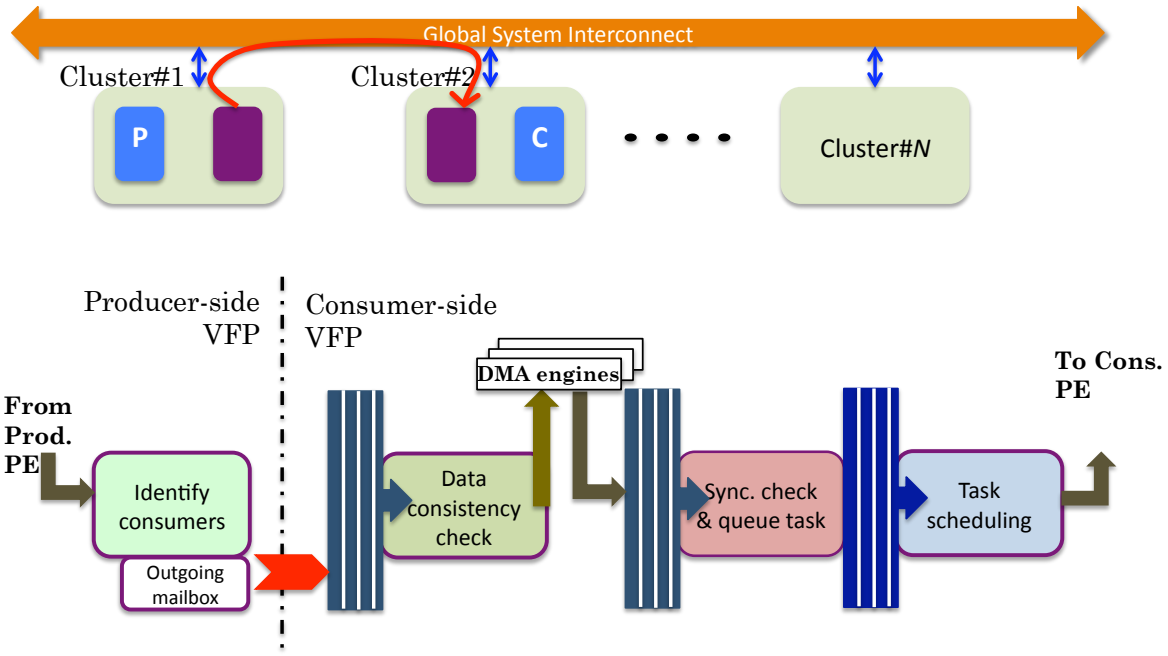


Figure 4.10: Inter-cluster Producer-Consumer Interaction

to gather control information from the data structures of the consumer, which lie in a remote cluster. This impacts performance because of considerably increasing the latency involved in activating the consumer task.

Having explained the reason for a primarily consumer-driven interaction, we now elaborate on how it is achieved for an inter-cluster scenario. The goal is to delegate the VFP control from the producer-side controller (that has identified the remote consumer) to the remote consumer-side controller. And to do this without disrupting or stalling the other PEs being served by either controller. So we need a scheme that is symmetric with respect to the *intra*-cluster producer-consumer interactions. This is achieved using message passing, as depicted in the Fig. 4.10 – from producer-side VFP to the consumer-side VFP. As seen in Fig. 4.10, this achieves maintaining the same nature of pipelined operation of the VFP controller – but across clusters. Thus, the distributed VFP controllers *share* (more by the consumer-side) the responsibility for managing inter-cluster producer consumer interactions.

Lets describe the stages involved at a slightly more detailed/microarchitecture level.

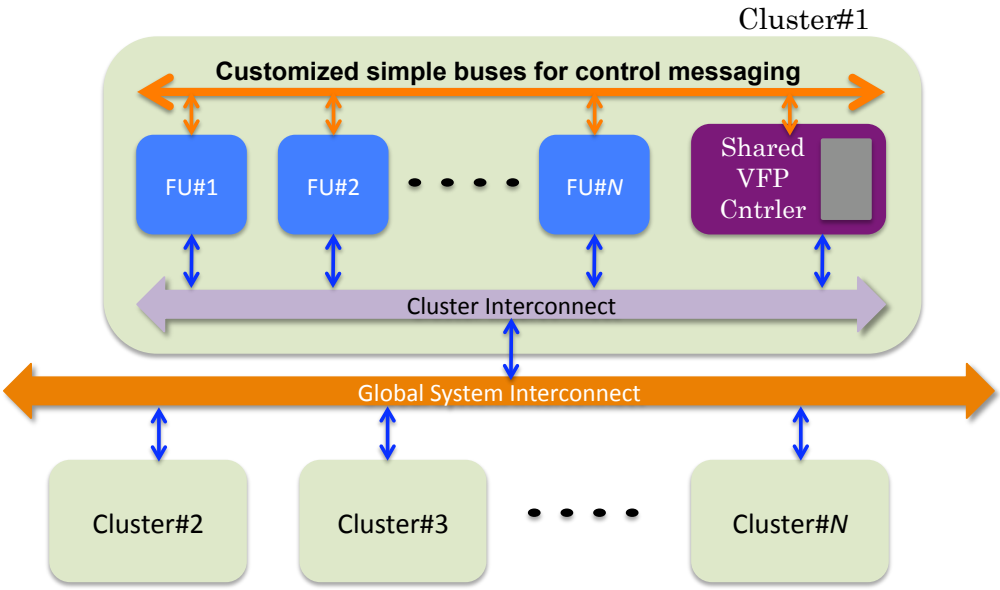


Figure 4.11: Complete Clustering-based SoC

An mentioned earlier, the scenario is identified by the Consumer identification stage at the producer-side, which then instead of writing the message into the Data consistency stage buffer, writes it into an outgoing mailbox. The outgoing mailbox controller detects this request to transfer an outgoing message. It then, via the DMA engine and global interconnect, transfers the message to the consumer-side Data consistency check stage buffer. The Data-consistency check stage at the consumer-side VFP controller then reads this message and continues the processing – oblivious to the fact that it is performing an inter-cluster interaction.

Thus, after all the considerations described in the previous sections, the final SoC architecture is as seen in Fig. 4.11. Next chapter evaluates and discusses the performance of the SoC architecture.

# Chapter 5

# Implementation and Performance Results

This chapter initially describes the implementation details of our design and also provides hardware complexity results. Further, it discusses the performance results for the clustering-based organization.

## 5.1 Implementation

As part of this work, the complete SoC platform (i.e. excluding the PEs) design and microarchitecture has been done at bit- and cycle- accurate level. Also, the design has been implemented in *synthesizable register-transfer-level* (RTL) using System Verilog Hardware Description Language.

The interconnect used in the system is the AMBA AXI 3.0 ([36]) based interconnect *DesignWare* IP from Synopsys[37]. The interconnect structure used is the *Multiple Address Multiple Data* structure – since Synopsys license we had could provide only this. Anyways, the problem of optimizing the interconnect, although impacts the performance, is orthogonal to this work.

The functional verification of the platform has been done using an *Open Verification Methodology* (OVM) based verification environment. The same environment is also used for measuring parameters (section 5.2) for evaluating the performance of the system. The details of the environment and functional verification can be found in [39].

### 5.1.1 Hardware Complexity

The complete VFP platform for a cluster of 8 PEs was synthesized using Synopsys *Design Compiler*[38]. The platform included the shared VFP controller and supporting distributed logic (i.e. context switch logic, FU messaging interface and DMA engines)

but no PEs. The resultant gate count for the platform is $114K$ gates. Gate count here is in terms of the NAND2 gates.

The gate count for the distributed VFP design was $25K$ gates per VFP controller. Hence, for 8 PEs the total VFP logic gate count becomes $25K \times 8 = 200K$ gates. Thus, hardware saved is approximately, $(200K - 114K)/200K = 0.43$ i.e. 43%. These are significant savings in hardware, which will also translate to power savings. This work has not calculated power numbers due to unavailability of EDA tools for it.

## 5.2 Evaluation Metrics and Parameters

The RTL simulations of the platform design under the OVM based environment[39] is used for collecting data for performance evaluation. Before getting into the performance discussions, following is a description of the parameters/metrics considered for the performance evaluations.

### 5.2.1 PE Utilization

PE utilization is defined as the percentage of the total time that the PE is busy processing tasks. Maximizing the PE utilization is the primary goal of the VFP architecture. In fact, it is the very rationale behind offloading all the task-flow management to the dedicated VFP controller.

The complement of PE utilization is nothing but the VFP control overhead. Hence, % of VFP overhead = 100 − PE utilization For e.g., suppose three tasks, each requiring processing time of 40 cycles, are executed on a PE using the VFP based control. The total time taken to complete the processing of all the three tasks is measured to be 150 cycles. Then the PE utilization in this case is $(40 \times 3)/150$=0.8 i.e. 80%. And the VFP overhead is 20%.

The goal for efficient processing of wireless protocols is to have high utilization even for short processing tasks. Hence the PE utilization values are plotted for different task processing times.

### 5.2.2 VFP Controller Idle Time

The VFP controller idle time, as the name suggests, is the percentage of the total time that the VFP controller is idle. As discussed in section 4.4.3, the VFP controller microarchitecture consists of stages. The VFP idle time is calculated for each of these stages, but while evaluating/ploting the values, only the lowest value is used. The reason being that the lowest value the most constraining one with respect to how much time the VFP is idle.

The significance of the VFP idle time measurement is to get an insight into the number of PEs that one shared VFP controller can handle efficiently.

### 5.2.3 PE Throughput

This value depicts the number of tasks completed by a PE per second. These throughput values are assuming a normalized frequency of $100MHz$. Increasing the number of PEs $x$, will provide $x$ times the throughput. These numbers are useful for the protocol programmer, since they help in deciding the tasks sizes, required number of PEs etc.

## 5.3 Performance Results

In this section we discuss the results with synthetic workloads. The aim is to provide some information for understanding the limits (in worst and best case) and characterization of the design.

A dummy PE was modeled in System Verilog for creating different simulation scenarios. This dummy PE is called golden PE. The golden PE was modeled with its task processing time and output data size as variable, using parameters.

Multiple platforms were created with different number of gold PEs in a cluster, i.e., different number of PEs (i.e. 2, 4, 6, 8, 10, 12 and 14) sharing the same VFP controller. For each case, 2 virtual flows were programmed and simulated. Each of the flows, for every configuration of number of PEs per VFP controller, consisted of tasks (one for each PE) in a chain. For example, for the case of 4 PEs sharing a VFP controller, each of the 2 flows programmed were in the sequence $1 -> 2 -> 3 -> 4$. This kind of

fully dependent flows are a worst case scenario from the VFP controller point of view, because it has to perform all the VFP control mechanisms for each of task. Also, the processing time for task for each PE in the flow was set to be equal.

In order to get well averaged values for the utilization, idle time etc., 100 frames for each flow were simulated. Thus, a total of 200 frames were simulated, i.e., 200 tasks were required to be performed by each PE. Moreover, the VFP controller had to perform all the control mechanisms for 200 $\times$ $number\ of\ PEs$ times (e.g. 800 times for 4 PE cluster).

For each of the configurations of number of PEs sharing a VFP controller (i.e. 2, 4, 6, 8, 10, 12 and 14), simulations were run executing 200 frames with increasing values of task processing times.

### 5.3.1 PE Utilization − No Data Transfer

The above described setup and simulations were performed, by setting the size of the output data of each of the PEs to 1 word. By doing this we account for all the overheads involved except for data transfer time − since it is just one word and can be neglected. Evaluating these simulations without the impact of data transfer is important because without any data transfer only factors affecting the PE utilization are the VFP scheme overhead and the impact of sharing the VFP controller. Thus, the no data transfer case becomes a worst case from the VFP controller point of view.

The Fig. 5.1 shows the curves of PE utilization v/s Task processing times (in clock cycles) for increasing number of PEs sharing a VFP controller. The values of PE utilization are averaged across all the PEs in the respective configuration.

Following are the observations from the plot;

- On the whole it can be seen that the PE utilization increases from left to right, i.e., with increasing Task processing times.
- The PE utilization numbers are lower for larger number of PEs per VFP controller, but only up to distinct cut-off points. For example, the 8 PE case has lower utilization values than the 2,4 and 6 PE configurations, only for tasks shorter

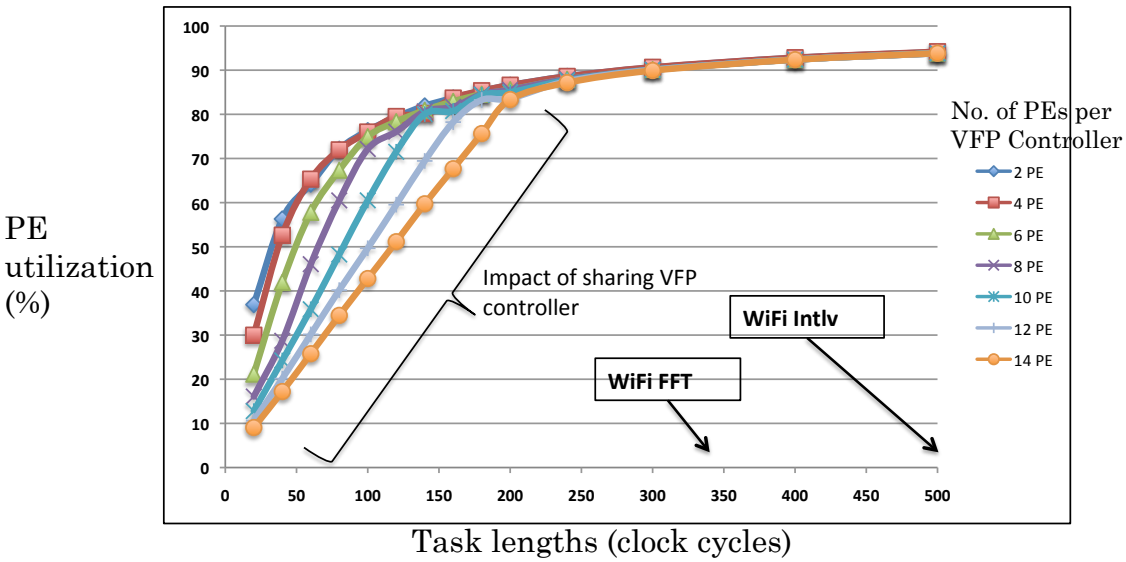


Figure 5.1: PE Utilization v/s Task Processing Time (No Data Transfer)

than 100 clock cycles. Above the 100 cycle task cutoff, the utilization is same as that for lower configurations. For the 14 PE case this cut-off value is task processing time of 200 cycles.

- These lower utilization numbers, up to a cutoff, for increasing number of PEs per cluster is due to the impact of sharing the VFP controller. Above the distinct cutoffs of task processing times, the impact of sharing the VFP is nullified. This is because for longer task processing times, the VFP controller has more idle time between requests by PEs. This claim is verified by the Fig. 5.2 discussed in the next section.

- Above the maximum cutoff of task processing time of 200 cycles, all the cluster configurations have a similar exponentially decreasing increase in PE utilization. This increase is primarily because the overhead of the VFP control scheme (excluding the sharing of VFP) is fixed; and once the impact of sharing the VFP is nullified, this fixed VFP overhead gets relatively smaller with increasing task processing times.

- The curves show that the VFP based architecture can attain high PE utilization (80% +), even with considerably short task processing times, e.g. 120 cycle tasks for 8 PE configuration. This is a good result since it enables the VFP based SoC to effectively support short processing tasks, e.g., FFT for five OFDM symbols (64 samples each) of 802.11a using a pipelined FFT architecture will need approximately 350 clock cycles (64 for FFT calculation and 6 for pilot insertion etc.). In comparison, the related work of *CoreManager*[13] and *OSIP*[40] only mention task lengths of up to 2000 clock cycles. This probably implies they cannot support lower task processing times effectively.

### 5.3.2 VFP Controller Idle Time

For the very same simulations as the previous section, curves of the VFP idle time are plotted against the Task processing times for the various configurations of PEs per clusters (i.e. 2, 4, 6, 8, 10, 12 and 14). These are depicted in Fig. 5.2.

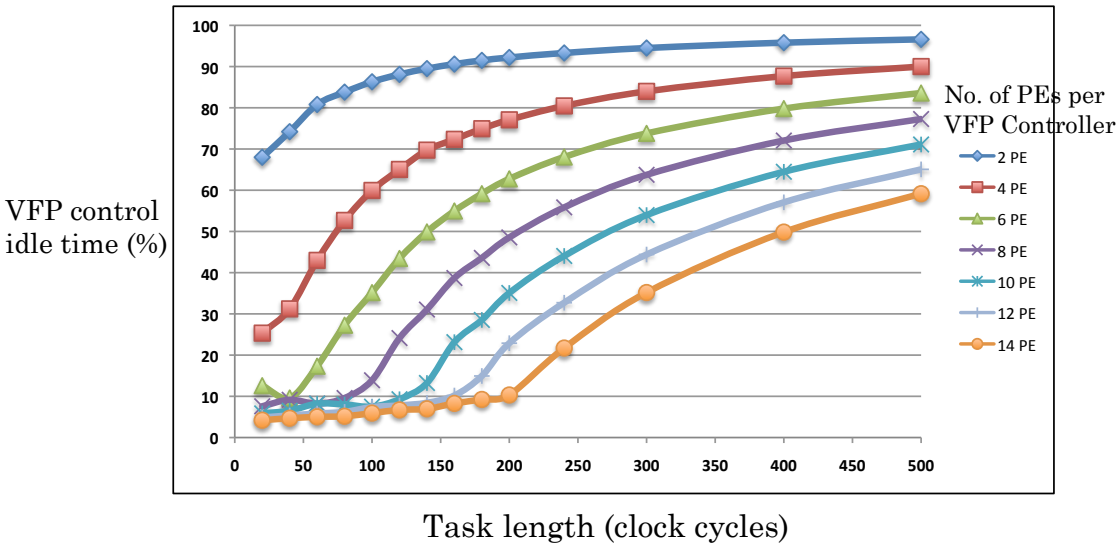


Figure 5.2: VFP Controller Idle Time v/s Task Processing Time

Lower values of VFP idle time signify that the VFP controller is kept busy – due

to many requests from the PEs. Values of under 10% indicate that the VFP controller is almost congested.

Following are the observations and deductions from Fig. 5.2;

- The VFP controller, confirming to intuitions, is increasingly busy for larger number of PEs per cluster.
- Clear cutoffs, in terms of task lengths, can be observed for different configurations indicating the minimum task sizes that each configuration can efficiently support. For example, for 8 PEs to share the VFP effectively, the task lengths must be longer than 100 clock cycles. Similarly, the minimum task length for the configuration of 14 PEs sharing a cluster is 200 clock cycles.
- It can be seen that these cutoffs are consistent with the cutoffs in the PE utilization curves discussed in the previous section. Thus proving our claims there.

The work in [40] present similar analysis of their work. Comparing the results shows that the VFP controller is more scalable, i.e., can handle more number of PEs while supporting short tasks, e.g., [40] needs minimum task sizes of 5000 cycles for supporting 8 PEs.

## 5.4 Impact of Data Transfer

This section discusses the impact of data transfer on the performance − specifically the PE utilization. Simulations similar to those in the previous section are run for different ratios of Data transfer time to Task processing time. These ratios are used instead of absolute data sizes or data transfer times because there is a strong correlation between the data transfer size and task processing time. Most of the operations/tasks of wireless protocol processing are very data dependent, e.g., pipelined FFT takes one clock cycle per sample. Also, the data transfer time will never really exceed the task processing time (assuming same clock frequency for both). Even if a PE task did no data operations and just wrote some output to the memory, the task processing time

will be equal to the time that it will take to transfer this data to a consumer's input. In this worst case the ratio of data transfer time to task processing time is 1.

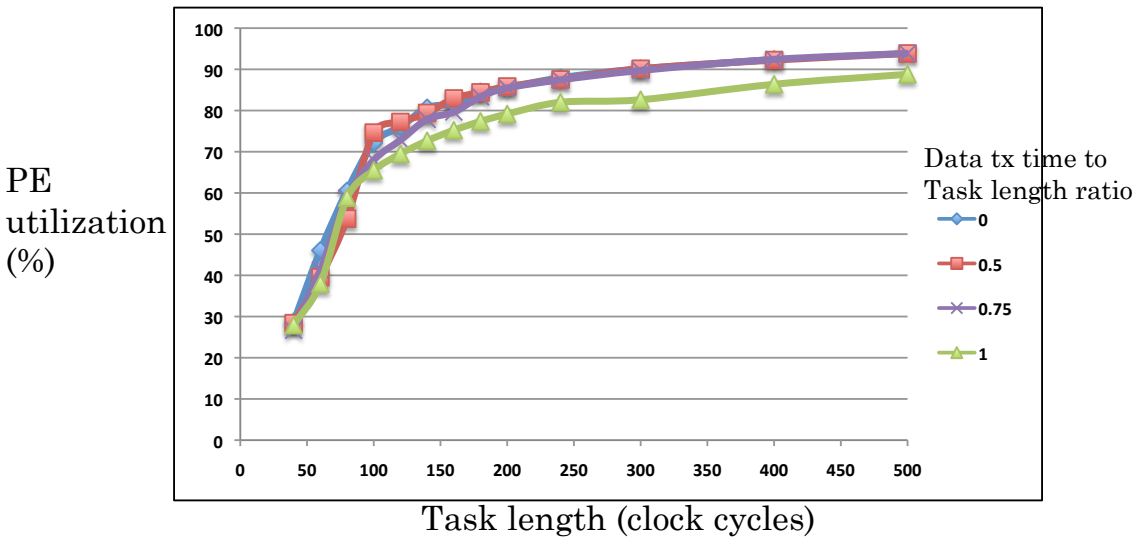


Figure 5.3: PE Utilization v/s Task Processing Time (Impact of Data Transfer)

The Fig. 5.3, shows the curves of PE utilization v/s Task processing time for different ratios (i.e. no data transfer, 0.5, 0.75 and 1) of Data transfer time to Task processing time. For clarity only curves for configuration of 8 PEs sharing a VFP controller are shown. The case of 8 PEs is of value because most of the wireless protocol flows will need up to 8 PEs for transmitter or receiver.

Observations from Fig. 5.3 are;

- The curve for the *no data transfer* case can be considered as the best case scenario with respect to the impact of data transfer.

- It can be seen that with increasing ratios of data transfer time to task processing time, the PE utilization remains unaffected till the case when data transfer time and task processing times become comparable, i.e., ratio is 1. The reason for this good or unaltered performance for lower ratios is that the VFP control mechanisms achieve a good overlap of processing a task and data transfer for the

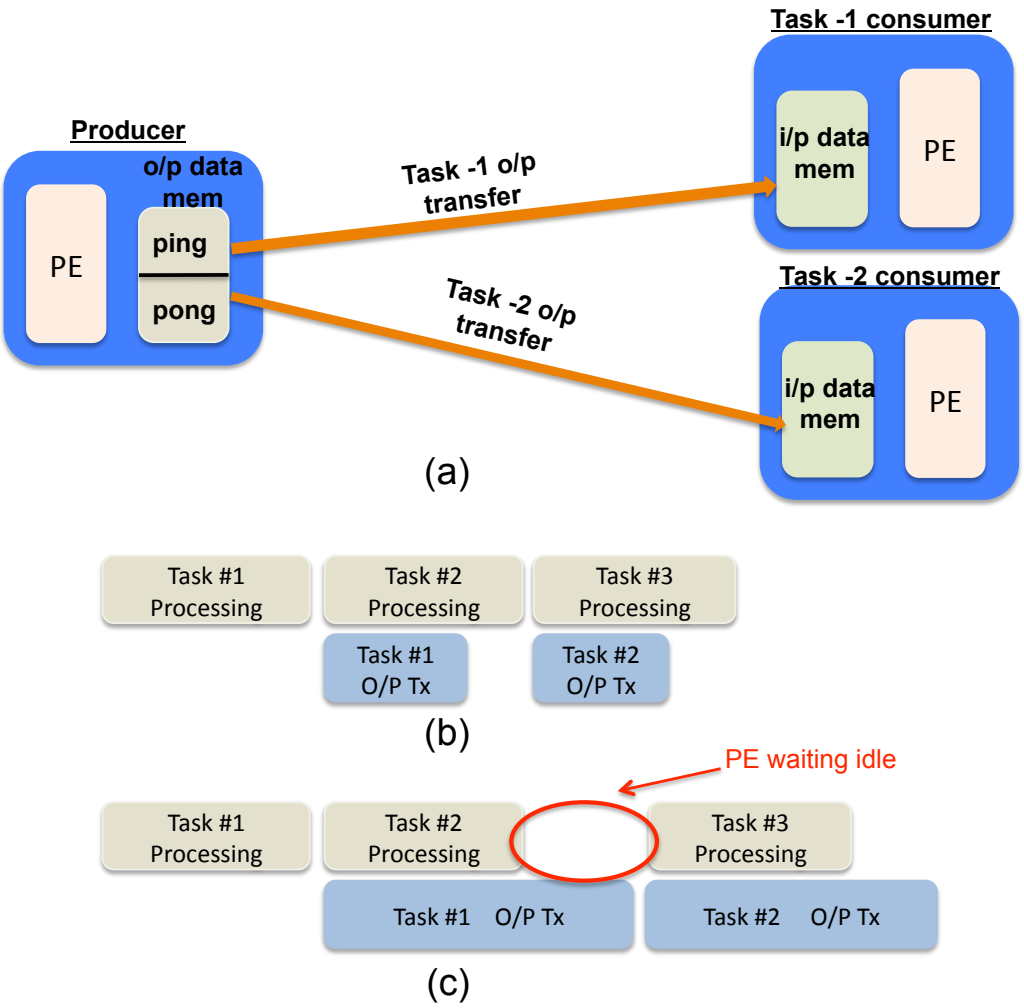


Figure 5.4: Impact of Data Transfer

previously completed task. The overlap is illustrated in Fig. 5.4b. This overlap is achieved using two regions (ping and pong) of output memory in an alternating fashion (Fig. 5.4a). The first task writes its output to the *ping* region. On completion of this task, the next task is immediately triggered and writes its output to the *pong* region. Concurrently data is transfered from the *ping* region to the consumer, thus achieving overlap and maximizing PE utilization by keeping it busy.

- For the case when data transfer time and task processing time are equal the PE utilization numbers show degradation. This is because the *ping-pong* scheme

discussed above, fails to keep the PE busy when data transfer takes longer than the PE processing. A task can be activated for a PE only if either *ping* or *pong* region is free, else PE remains idle. This is precisely what happens when the ratio is 1. As can be seen in Fig. 5.4c, the output data transfer of the first task (i.e. ping region) has not completed even beyond the completion of the second task (that has used *pong*). Hence, a new task cannot be immediately activated, resulting in the PE waiting idle that degrades utilization.

The sustained utilization numbers even for up to 75% ratio of data transfer time to task processing time, indicates the success of the architecture is trying to minimize the impact of data transfer on performance.

## 5.5 PE Throughput

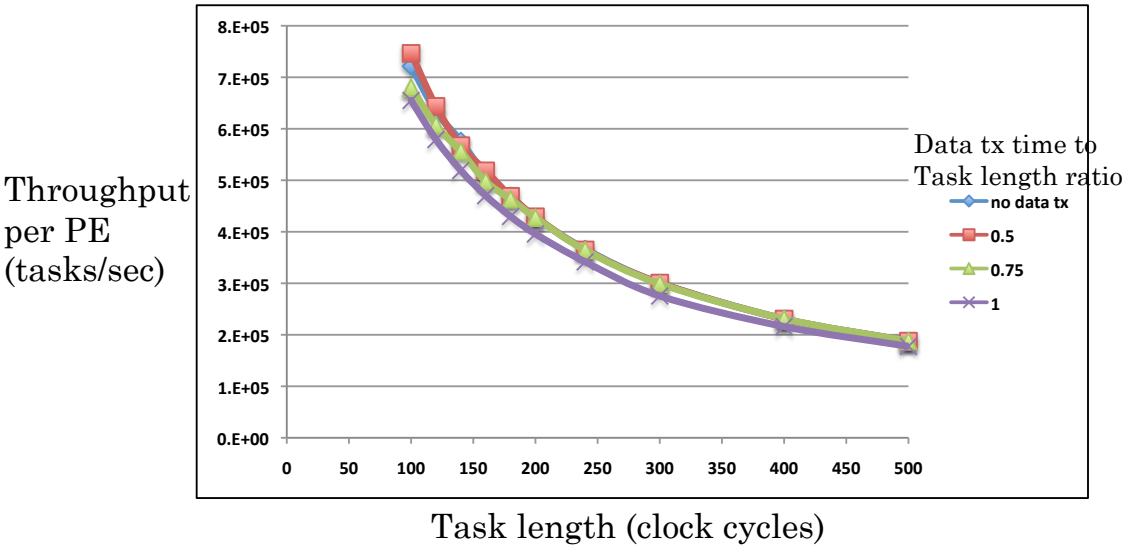


Figure 5.5: PE Throughput v/s Task Processing Time

The Fig. 5.5, plots the throughput per PE v/s task processing times. The throughput is the tasks completed per second. The curves are plotted for 8 PE configuration and different ratios of data transfer time to task length. It can be seen that one PE can

achieve a throughput of $200K\ tasks/second$ for tasks length of 500 clock cycles (i.e. $5\mu s$ task length).

## 5.6 Real Life Application – 802.11a

In order to have a more realistic benchmarking, the platform designed as part of this work has been integrated with the PEs required for a 802.11$a$-like transmitter. This work is detailed in [8].

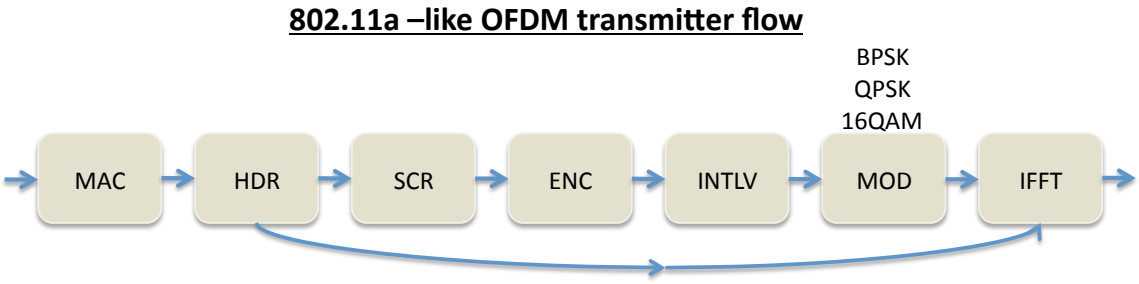


Figure 5.6: Real Life Application – 802.11a - like OFDM Transmitter

The constituent PEs and their flow is depicted in the Fig. 5.6. The constituent PEs are;

**MAC** : This PE performs the tasks of a reconfigurable MAC and currently supports two protocols ALOHA and CSMA-CA back off. It also uses 802.11a compatible inter-frame spacing (IFS) durations and fame formats.

**Header (HDR)** : This engine performs the task of appending the header as per the 802.11a frame format to the frame sent by the MAC engine. It also forks another task to the IFFT PE to initiate the transmission of the 802.11a packet preamble. This way the preamble is just transmitted by the time the actual data frame arrives at the IFFT for transmission. The HDR PE also performs the chunking of the frame so that PEs further down the flow operate on smaller data sizes, thus reducing overall latency.

**Scrambler (SCR)** : This PE is the length-127 scrambler with its polynomial ($x^7 + x^4 + 1$) as per the 802.11a standard.

**Encoder (ENC)** : This is the rate 1/2 convolution encoder required for 802.11a data rates of 6, 12 and 24 *Mbps*.

**Interleaver (INTLV)** : The PE performing the interleaving function as per the 802.11a standard.

**Modulator (MOD)** : This is the modulator (mapper) that can support modulation schemes of $BPSK$, $QPSK$ and $16QAM$ needed for the 802.11a data rates of 6, 12 and 24 *Mbps*.

**IFFT** : This PE is responsible for performing the IFFT function on the data from the modulator along with pilot insertion and cyclic insertion in order to create OFDM symbols for transmission as per the 802.11a standard. The symbol size for 802.11a is 64, hence the PE calculates 64 point IFFT. As mentioned previously, this PE also generates the packet preamble.

Detailed results and experiments are described in [8]. Here some of the key results are presented.

- The platform successfully supported 802.11a data rates of 6, 12 and 24 *Mbps*. These were the only rates that could be simulated because the Encoder PE could do only rate 1/2 encoding. Table 5.6 shows the peak throughput values, i.e., the system provides the output at the peak rate but a rate matching FIFO is used after the IFFT PE to transmit at the frequency satisfying the 802.11a data rate (6, 12 and 24 *Mbps* respectively.)

- The platform also supported two simultaneous flows at a time, for each data rate, i.e., 6, 12 and 24 *Mbps*. Table 5.6 provides the peak throughput numbers. The 2 flows of 24 Mbps can be output at a peak rate of 88.5 *Mbps*. It can be seen that the peak throughput increases by less than twice when the 802.11a data rate doubles. This is because the PEs after the MAC operate on larger data sizes

Table 5.1: Throughput Results for 802.11a-like OFDM Flows

| **Target Rate (Mbps)** | **Peak Throughput Measured for Single Flow (Mbps)** | **Peak Throughput Measured for Two Flows (Mbps)** |
|---|---|---|
| 6 | 7.47 | 26.98 |
| 12 | 16.25 | 51.06 |
| 24 | 34 | 88.5 |

for higher data rates (assuming they work on same number of OFDM symbols for each data rate). The high peak throughput demonstrates the potential of the platform to support more than 2 concurrent flows, but could not be simulated due to implementation restrictions. The restriction is that the IFFT PE is a legacy design which has maximum of 2 FIFOs to receive its output. Forcing higher number of flows breaks the simulations due to errors.

# Chapter 6

# Conclusion and Future Work

This work designed and implemented a clustering-based SoC architecture based on the hardware-oriented *Virtual Flow Pipelining* framework for programmable radio processing. A task manager (VFP controller), which can be shared across multiple PEs, is also designed to implement OS-like mechanisms in hardware. Based on the implementation and performance results discussed in chapter 5 we can have the following conclusions;

- The results prove the performance advantage of the hardware-based control implemented using the VFP framework. The platform is capable of efficiently (with PE utilization numbers of 80+%) for short tasks (i.e. task length in hundreds of clock cycles), as opposed to other related work ([13, 40]) that support task lengths above 2000 cycles. This is an important result because wireless protocols demand support for short tasks.

- The parallelism of task processing and data transfers, exploited by the implemented platform, achieves minimizing the impact of data transfer on performance. This result is important because the wireless protocol workload is very IO/data intensive.

- The asynchronous pipelined architecture of the shared VFP controller is very scalable. The design has successfully minimized the performance impact of sharing the control, e.g., even 14 PEs sharing the controller can achieve 80+% utilization for task lengths of 250 clock cycles.

- The shared VFP controller has provided significant hardware saving, thus enabling the clustering-based SoC organization that provides a balance between the scalability and hardware overhead. A cluster size of 8 PEs is a good choice,

because it can efficiently support (with 80+% utilization) tasks as small as 120 clock cycles. The hardware saving are also significant (43%) providing a solution with reasonable hardware overhead. Moreover, many of the wireless protocol flows need up to 8 PEs, thus minimizing the number of high latency inter-cluster producer-consumer interactions.

### Future Work

The following points can be considered for improving this work and developing on it;

- Including the feature of *speculative* data transfer. In the current architecture, the transfer of data happens only after the the consumer has been identified, which is required because the actual consumer can depend on the result. This is equivalent to having an *if-else* control dependency, e.g. the consumer of a CRC check could be different depending on whether the check passes or fails. But this sequence of actions adds to the system latency. Speculating the consumer will enable initiating the data transfer earlier (in fact in parallel with the task processing), thus reducing latency. A mechanism is required to identify the actual consumer and then take corrective measures if the speculation fails. This speculative scheme has good potential for performance improvement because such control dependencies in wireless protocols are rare.

- Estimating power numbers and introducing power-control features. A weakness of this work is that it does not estimate the power requirement of the platform. This was due to the unavailability of EDA tools. But it is an important analysis that should be performed. Also, the VFP controller must be enabled with features to control the voltage and frequency of PEs in a dynamic manner. This will help achieve a power efficient solution − vital for a wireless solution.

- Performing FPGA-based validation. The implementation and execution of this platform on a multi-board, multi-chip FPGA-based hardware can build even more confidence regarding its capabilities.

- Implementing more real life wireless protocols. More number of real wireless protocols (e.g. WiMax, etc.) must be programmed and executed on the platform. This will help gaining insights on the impact of real workloads and their diversity, thus exposing more areas for improvement.

# References


[1] What Is Software Defined Radio? www.wirelessinnovation.org/page/What_is_SDR.

[2] SDR Forum: Cognitive Radio Definitions. www.sdrforum.org/pages/documentLibrary/documents/SDRF-06-R-0011-V1_0_0.pdf.

[3] Frank Rayal. Infrastructure Design: Programmable Devices Vs. ASIC. www.wirelessweek.com/Articles/2010/04/Devices-Infrastructure-Design-Programmable-Vs-ASIC/.

[4] Zoran Miljanić, Ivan Seskar, Khanh Le, and Dipankar Raychaudhuri. The WINLAB network centric cognitive radio hardware platform: WiNC2R. *Mob. Netw. Appl.*, 13(5):533–541, 2008.

[5] J. Mitola. *Cognitive radio: An Integrated Agent Architecture for Software Defined Radio*. PhD thesis, KTH Royal Institute of Technology, 2000.

[6] Dake Liu, Anders Nilsson, Eric Tell, Di Wu, and Johan Eilert. Bridging Dream and Reality: Programmable Baseband Processors for Software-defined Radio. *Comm. Mag.*, 47(9):134–140, 2009.

[7] Zoran Miljanić and Predrag Spasojević. Resource Virtualization with Programmable Radio Processing Platform. In *WICON '08: Proceedings of the 4th Annual International Conference on Wireless Internet*, pages 1–7, ICST, Brussels, Belgium, Belgium, 2008. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[8] Madhura Joshi. System Integration and Performance Evaluation of WiNC2R Platform for 802.11a-like Protocol. Master's thesis, Rutgers University, October 2010.

[9] J. Martin and F. Clermidy. Mobile Digital Baseband: From Configurable to Evolutive Platforms. In *Mobile and Wireless Communications Summit, 2007. 16th IST*, pages 1 –5, 1-5 2007.

[10] A. Sangiovanni-Vincentelli, L. Carloni, F. De Bernardinis, and M. Sgroi. Benefits and Challenges for Platform-based Design. In *Design Automation Conference, 2004. Proceedings. 41st*, pages 409 – 414, 2004.

[11] C. H. van Berkel. Multi-core for Mobile Phones. In *Design, Automation and Test in Europe, DATE 2009, Nice, France*, pages 1260–1265. IEEE, April 20-24, 2009.

[12] G. Desoli and E. Filippi. An Outlook on the Evolution of Mobile Terminals: from Monolithic to Modular Multiradio, Multiapplication Platforms. *Circuits and Systems Magazine, IEEE*, 6(2):17 –29, 2006.

[13] T. Limberg et. al. A Heterogeneous MPSoC with Hardware Supported Dynamic Task Scheduling for Software Defined Radio. In *In Proceedings of the 46th Design Automation Conference (DAC'09)*, July 26-31 2009.

[14] Liesbet Van der Perre, Jan Craninckx, and Antoine Dejonghe. SDR Baseband Platforms. In *Green Software Defined Radios*, Integrated Circuits and Systems, pages 65–95. Springer Netherlands, 2009.

[15] David Patterson. The Parallel Computing Landscape: a Berkeley View. In *Low Power Electronics and Design (ISLPED), 2007 ACM/IEEE International Symposium on*, pages 231 –231, 27-29 2007.

[16] Qiwei Zhang, Andre B.J. Kokkeler, and Gerard J.M. Smit. Cognitive Radio Design on an MPSoC Reconfigurable Platform. In *Cognitive Radio Oriented Wireless Networks and Communications, 2007. CrownCom 2007. 2nd International Conference on*, pages 187 –191, 1-3 2007.

[17] Dominique Nussbaum, Karim Kalfallah, Christophe Moy, Amor Nafkha, Pierre Lerary, Julien Delorme, Jacques Palicot, Jérôme Martin, Fabien Clermidy, Bertrand Mercier, and Renaud Pacalet. Open Platform for Prototyping of Advanced Software Defined Radio and Cognitive Radio Techniques. In *DSD '09: Proceedings of the 2009 12th Euromicro Conference on Digital System Design, Architectures, Methods and Tools*, pages 435–440, Washington, DC, USA, 2009. IEEE Computer Society.

[18] Kees van Berkel, David van Kampen, Orlando Moreira, Petr Kourzanov, Marinus Splunter, Antti Piipponen, Kalle Raiskila, Sverre Slotte, and Tommi Zetterman. Multiradio Scheduling and Resource Sharing on a Software Defined Radio Computing Platform. In *SDR'09 Technical Conference and Product Exposition*, Dec 1-4, 2009.

[19] Liesbet Van der Perre, Wim Van Thillo, Antoine Dejonghe, and Joris Van Driessche. Tomorrows Wireless Communication Requires Higher Throughput and a Smaller Energy Budget. *Chip Design Magazine*, pages 34 –38, June/July 2010.

[20] Olli Silven and Kari Jyrkkä. Observations on Power-efficiency Trends in Mobile Communication Devices. *EURASIP J. Embedded Syst.*, 2007(1):17–17, 2007.

[21] T. Limberg, M. Winter, M. Bimberg, R. Klemm, E. Matus, M.B.S. Tavares, G. Fettweis, H. Ahlendorf, and P. Robelly. A Fully Programmable 40 GOPS SDR Single Chip Baseband for LTE/WiMAX Terminals. In *34th European Solid-State Circuits Conference, 2008. ESSCIRC 2008.*, pages 466 – 469, Sept 5-18, 2008.

[22] O. Arnold and G. Fettweis. Power Aware Heterogeneous MPSoC with Dynamic Task Scheduling and Increased Data Locality for Multiple Applications. In *in Proceedings of the X International Workshop on Systems, Architectures, MOdeling, and Simulation (SAMOS'10)*, 19-22 July 2010.

[23] Camille Jalier, Didier Lattard, Gilles Sassatelli, Pascal Benoit, and Lionel Torres. Flexible and Distributed Real-time Control on a 4G Telecom MPSoC. In *Proceedings of 2010 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 3961 – 3964, 2010.

[24] IMEC COBRA: Imecs cognitive baseband radio to support 4G and broadband access to multiple services. http://www2.imec.be/be_en/press/imec-news/cobra.html.

[25] D. Pulley. Multi-core DSP for Basestations: Large and Small. In *Design Automation Conference, 2008. ASPDAC 2008. Asia and South Pacific*, pages 389 –391, 21-24 2008.

[26] U. Ramacher. Software-Defined Radio Prospects for Multistandard Mobile Phones. *Computer*, 40(10):62 –69, oct. 2007.

[27] John Glossner, Daniel Iancu, Mayan Moudgill, Gary Nacer, Sanjay Jinturkar, Stuart Stanley, and Michael Schulte. The Sandbridge SB3011 Platform. *EURASIP J. Embedded Syst.*, 2007(1):16–16, 2007.

[28] Yuan Lin, Hyunseok Lee, M. Who, Y. Harel, S. Mahlke, T. Mudge, C. Chakrabarti, and K. Flautner. SODA: A Low-power Architecture For Software Radio. In *Computer Architecture, 2006. ISCA '06. 33rd International Symposium on*, pages 89 –101, 0-0 2006.

[29] Hyunseok Lee. *A Baseband Processor for Software Defined Radio Terminals.* PhD thesis, University of Michigan, Ann Arbor, MI, USA, 2007. Adviser-Mudge, Trevor N.

[30] Najam-ul-Islam Muhammad, Rizwan Rasheed, Renaud Pacalet, Raymond Knopp, and Karim Khalfallah. Flexible Baseband Architectures for Future Wireless Systems. In *DSD '08: Proceedings of the 2008 11th EUROMICRO Conference on Digital System Design Architectures, Methods and Tools*, pages 39–46, Washington, DC, USA, 2008. IEEE Computer Society.

[31] Zoran Miljanic and Khanh Le. Processing Engine Interface Specification. www.svn.winlab.rutgers.edu/cognitive/Architecture/r1/pe.

[32] Onkar Sarode. Centralized VFP Controller Architecture Document. www.svn.winlab.rutgers.edu/cognitive/Architecture/r2/vfp.

[33] Onkar Sarode. Centralized VFP Controller Architecture Visio Diagrams. www.svn.winlab.rutgers.edu/cognitive/Architecture/r2/vfp.

[34] Onkar Sarode. Functional Unit Interface and Architecture Document. www.svn.winlab.rutgers.edu/cognitive/Architecture/r2/fu.

[35] Onkar Sarode. Functional Unit Interface and Architecture Visio Diagrams. www.svn.winlab.rutgers.edu/cognitive/Architecture/r2/fu.

[36] AMBA 3.0 AXI. www.arm.com.

[37] DesignWare IP Solutions for the AMBA Interconnect. http://www.synopsys.com/dw.

[38] Synopsys Design Compiler User Guide. http://www.synopsys.com/Tools.

[39] Akshay Jog. Architecture Validation of the WiNC2R Platform. Master's thesis, Rutgers University, October 2010.

[40] J. Castrillon, Diandian Zhang, T. Kempf, B. Vanthournout, R. Leupers, and G. Ascheid. Task Management in MPSoCs: An ASIP Approach. pages 587 –594, nov. 2009.