Università degli Studi di Cagliari

DOTTORATO DI RICERCA

in

Ingegneria Elettronica e dell'Informazione

Ciclo XXXI

TITOLO TESI

Power and Energy Management in Coarse-Grained Reconfigurable Systems: methodologies, automation and assessments

Settore scientifico disciplinari di afferenza

SSD 09 / ING-INF/01

Presentata da: Tiziana Fanni

Coordinatore Dottorato: Prof. Fabio Roli

Tutor: Prof. Luigi Raffo

Co-Tutor: Dott.ssa Francesca Palumbo

Esame finale anno accademico 2017-2018

Tesi discussa nella sessione d'esame Febbraio 2019

Ph.D. in Electronic and Computer Engineering
Dept. of Electrical and Electronic Engineering
University of Cagliari

Power and Energy Management in Coarse-Grained Reconfigurable Systems: methodologies, automation and assessments

Tiziana Fanni

Advisor: Prof. Luigi Raffo
Co-Advisor: Dr. Francesca Palumbo
Ph.D. Coordinator: Prof. Fabio Roli
Curriculum: ING-INF/01

XXXI Cycle
A.A. 2017-2018
4th February 2019

Abstract

In the era of Cyber-Physical Systems (CPS), designers need to cope with several constraints that have to be met at the same time. CPS are complex systems composed of different interactive and deeply intertwined components that have to change their behavioural modalities according to several factors as the environment status, requests from user and even their internal status, thus requiring high flexibility and performance, possibly with a low power consumption. The spectrum of existing computing systems ranges from general purpose to application specific systems. General purpose systems as CPUs, GPUs, DSPs offer high flexibility but are not able to provide high performance, due to their poor specialization. On the other side, Application Specific Integrated Circuits (ASICs) offer high performance but they do not provide flexibility at all, being designed for computing a single, specific application. In the middle between general purpose systems and ASICs lie the reconfigurable systems that provide a valuable solution to challenge simultaneously different requirements. Reconfigurable systems offer a certain level of flexibility, while guaranteeing high performance. However, two major issues still limit their wide applicability: high design complexity, implying huge engineering effort, as well as power inefficiencies.

The activities behind my thesis address both these issues, with the primary focus on power consumption. The starting assumption is the definition of a set of strategies that, depending on the considered scenario and the chosen target device (ASIC or FPGA), may enable power/energy awareness and consumption optimization. In parallel, these strategies have been automated within different extensions of a dataflow to hardware design suite for coarse-grained reconfigurable systems.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

We are in the Cyber-Physical Systems (CPS) era and designers need to offer support for advanced adaptivity. CPS are complex systems composed of different interactive components, which need to meet several requirements imposed by the environment, the user and even their internal status. In particular adaptivity triggers are classified as:

- functional-oriented: adaptation needed to offer different functionalities over the same substrate or to maintain correct functionality, e.g., because the CPS mission changes, several functions running on the same HW interchangeably, or the data being processed changed and adaptation is required. It may be parametric (e.g., a weights or parameters changes) or fully functional (e.g., the algorithm changes). For example, it might be necessary to change the type of filtering according to the type of noise in order to provide the required functionality.

- non-functional-oriented: functionality is fixed, but system requires adaptation caused by non-functional requirements, such as performance or available energy. For example, filtering precision could be reduced in case of low battery.

- repair-oriented: for safety and reliability purposes, adaptation may be used in case of faults. Adaptation may add self-healing or self-repair features. For example, HW task migration for permanent faults, or scrubbing (continuous fault verification) and repair.

Reconfigurable systems provide a valuable solution to offer support for adaptivity: lying in the middle between general purpose computing platforms and application-specific circuits, they offer tradeoffs between flexibility, performance and power efficiency. In this thesis, coarse grained reconfigurable (CGR) systems have been addressed. In CGR systems, all the resources belonging to all the possible configurations are always instantiated in the substrate, and reconfiguration is achieved by multiplexing the resources in time, switching among configurations. The outcome is high-performance and fast reconfiguration, while the main drawbacks are related to: (1) maximum operating frequency, which is less than or equal to the one of the isolated configurations; (2) area utilization, due to the presence of resources that are not involved in the configuration that is active; (3) flexibility, being able to compute only the functionalities the system has been designed for. This makes coarse-grain

reconfiguration suitable for application specific real-time contexts, where only few different behaviours are required. CGR solutions can be adopted either in Application Specific Integrated Circuits (ASIC), incrementing the intrinsic flexibility of this high performance circuitry, or in Field Programmable Gate Array (FPGA) architectures, to provide a further level of flexibility besides the native fine-grain one. In particular, in the FPGA case the CGR can be combined with the dynamic partial reconfiguration (DPR) to achieve very high flexible and adaptable chips capable of switching, at runtime, among different sets of functionalities and working points. The main issue of CGR devices is the complexity of their design under several aspects: resources mapping, optimisation, hardware design, runtime management. Moreover, reconfigurability alone is still unable to solve the power issue.

In the last decade, the adoption of model-based design by electronic systems designers has increased. System-level modelling enables early stage analysis and optimisation, helping in meeting and verifying the design constraints. In particular dataflow models of computation present an intrinsic modularity that is natively suitable to manage execution concurrency typical of multi-processor environments. Furthermore, they can be exploited to facilitate system development, being possible to map dataflow entities to hardware blocks. For these distinctive features, dataflow models of computation together with coarse-grain reconfiguration, constitute the pillar of the approach adopted in this thesis.

1.1 Objectives of the Thesis

The main objective of this thesis work is the development of automated methodologies for the design and management of low power CGR systems. The methodologies, leveraging on the dataflow models of computation, will be able to provide power/energy awareness and consumption optimization depending for different application scenarios and technology target (ASIC or FPGA). These methodologies have been automated within different extensions of the Multi-Dataflow Composer Tool, a dataflow to hardware design suite for CGR systems. In particular this main objective, can be divided in the following ones:

- different approaches for power/energy saving strategies will be explored, to provide power awareness and optimization for different target devices (ASIC and FPGA).
- system-level modelling is crucial in the design of CGR systems. A dataflow-driven methodology able to help designers to experiment with new optimization techniques for dataflow-based implementations will be studied.
- the methodologies have to be automated, exploiting already existent dataflow oriented tools, so that all the steps of the reconfigurable systems design flow (resource mapping, optimisation, hardware design and runtime management) will be supported.

1.2 Thesis Structure

The thesis structure firstly provides in Chapter 2 a brief state of the art of the concepts at the basis of the proposed approach: reconfigurable computing, dataflow models of computation and power issue. Chapter 3 present an overview of the Multi-Dataflow Composer

tool, which is the main tool exploited in the work of this thesis, to automate the implementation of the different power management techniques in dataflow-based CGR systems that are explored in this thesis, and presented in the following Chapters:

- Chapter 4 will discuss power management in ASIC systems, and will present a methodology for the automatic implementation of a power gating strategy to CGR systems;
- Chapter 5 will extend the implementation of power saving strategies for CGR systems, dealing with a power modelling methodology capable of determining, prior to any physical system implementation, the cost of the saving strategy and its best implementation;
- Chapter 6 will propose a dataflow-driven methodology capable of helping the designers to rapidly incorporate and efficiently experiment with new optimization techniques for dataflow-based implementations;
- Chapter 7 will describe and automated framework for the development and runtime management of multi-grain reconfigurable hardware systems, able to to provide different tradeoffs between performance, flexibility and energy consumption, reaching the advanced runtime adaptivity support necessary for CPS.

Lastly, Chapter 8 will report some considerations about the thesis work, going down in the detail for all the presented methodologies and giving some directives for future improvements.

Chapter 2

The Context of Low Energy Reconfigurable Digital Circuit Design for Streaming applications

In literature there is a wide variety of processing systems that provide different degrees of performance and flexibility. Figure 2.1 depicts a graphical overview of how the existing processing systems are spread between the axes representing these two metrics. Closed to the flexibility axis there are general purpose systems, such as the Central Processing Units (CPUs) commonly adopted in personal computers. These devices are able to execute a huge quantity of functionalities, since they support different programming languages that, through proper compilers, are translated in machine code. The machine code is the sequence of instruction calls of the CPU that leads to the execution of a specific functionality. The instructions supported by CPUs are generic operations, like the transfer of a data from the main memory to one of the processor registers or a simple addition between the values contained onto two registers. They are not conceived for a particular application. Actually, by means of programming languages the user can model its custom applications with a degree of flexibility that is the maximum achievable for an electronic device.

Leaving the flexibility axis, there are some devices that can be seen as general purpose ones, but that are more suitable for a specific kind of applications. Examples of this category of devices are Graphical Processing Units (GPUs) and Digital Signal Processors (DSPs). The GPUs are processing systems dedicated to the execution of tasks related to the image and video processing. Their architecture is designed for the exploitation of the data parallelism that typically characterises this kind of applications. Despite GPUs are conceived for a specific class of operations, they are able to provide a huge variety of functionalities due to their support for several programming languages. However, the best performance is achievable only with the image and video processing applications and/or if the application is written in order to optimally exploit the GPU architecture. At this purpose several API and frameworks aiming at the usage of GPUs for general purposes (GPGPUs) by modifying, automatically or not, the generic code of a given application have been proposed in literature (e.g the CUDA architecture from NVIDIA [77] or the OpenCL library [2]).

DSPs are another kind of almost general purpose devices that provide specific machine instructions for signal processing operations. The most common additional instructions

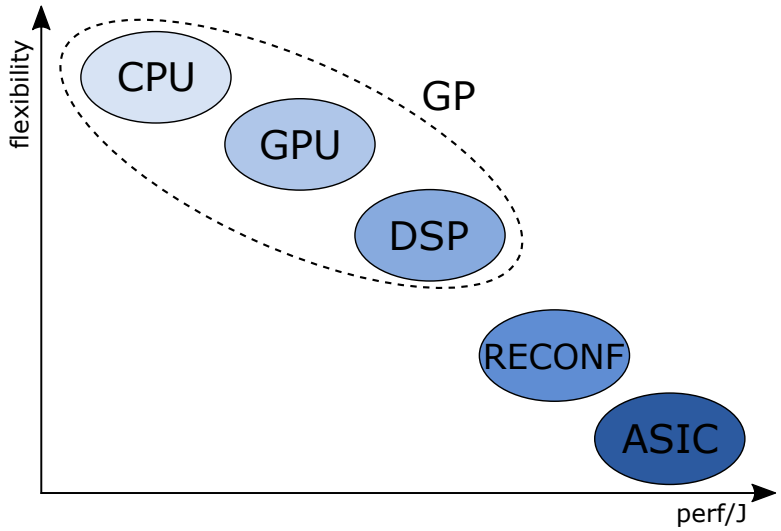


Figure 2.1: Flexibility versus performance graph of the main architectural alternatives for processing systems.

provided by DSPs are multiply and accumulate, typically useful in matrix operations, filtering and transforms. Very often DSPs natively support fixed and floating point arithmetic through dedicated logic units, thus speeding-up the operations involving these number formats. Furthermore, the DSP architectures can also be able to exploit data parallelism that may be present in the signal processing applications. Like for GPUs, DSPs have dedicated compilers to support programming languages and, theoretically, to run any kind of program, but they achieve the best performance only if the executed code contains common signal processing operations and/or if it is written in order to fully exploit the system potentials (DSP primitives may be necessary).

All the mentioned devices, CPUs, GPUs and DSPs, can be grouped within the general purpose systems since, by means of programming languages, they can execute any kind of application. However, for the same reason, they are not uniquely designed to perform a specific task. The execution of a single instruction in general purpose systems is composed of different stages: instruction fetch, instruction decode, execution, memory and write back. Each stage can last one or more clock cycles and an application can be composed of several thousands of instructions. Even though the standard execution stages can be pipelined or replicated, it is straightforward that this is not the most efficient way to execute a given functionality. To push the performance, the design approach has to be reversed: if for the general purpose systems the wanted functionalities are adapted to a prefixed architecture, in maximum performance systems the architecture has to be adapted to the wanted functionality. This is what occurs for the Application Specific Integrated Circuits (ASICs) on the opposite side of the flexibility versus performance graph of Figure 2.1. ASICs, as the same acronym reveals, are systems that are designed exclusively for the execution of a given functionality. All and only the involved resources are needed to perform the functionality and the architecture is forged accordingly to the native execution flow of the application. Obviously, the design can be affected by other constraints, such as the minimisation of the consumed power, besides the performance maximisation. This may lead to different versions of ASICs for a given application. ASICs are not flexible at all because no other operations can be performed, apart the one around which the system has been built.

A huge difference between the adoption of general purpose systems or ASICs for the execution of applications stands on the designer effort, costs and required skills. Dealing with general purpose systems the designer has firstly to choose the target architecture among a wide variety of available devices (also cheap solutions are possible), and then to shape its code accordingly. He has not to worry about the system architecture design, so hardware designing skills are not necessary. Dealing with ASICs is different, since the architecture itself has to be characterized, thus requiring very specific hardware designing expertise. The development of ASICs is generally a longer, more expensive and more complex process than adopting general purpose systems. Additional choices are possible between the two described extreme points, general purpose systems and ASICs, of the flexibility versus performance graph in Figure 2.1. An appealing option, constituting a trade-off among the previous systems, is given by reconfigurable computing, that offer tradeoff between performance and flexibility.

However, dealing with modern embedded devices, one crucial aspect to be taken into consideration is the system consumption. As a matter of fact, the portable era has raised the problem of power dissipation to the first positions of the design constraints. Battery life limits force devices to exploit each single energy particle in the maximum efficient way, so that a dedicated power management strategy usually needs to be defined within the project of modern embedded systems.

In order to overcome the problems of low-power reconfigurable systems development and evolution, the present work exploits the dataflow paradigm. Dataflows own distinctive features, like modularity and parallelism highlighting, that may help designers targeting a particular class of reconfigurable systems. A detailed description of dataflow paradigm and of the DPS-oriented dataflow Models of Computation is provided by Section 2.2.

The rest of the chapter is organised as follow: Section 2.1 will describe the main aspects of reconfigurable computing and will show a survey of the main reconfigurable architectures proposed in literature. Section 2.2 will present the dataflow paradigm and the particular formalism that has been adopted in this thesis work. Section 2.3 involves an overview on the actual power issue in digital systems. Because of the heterogeneity of this thesis, that explores different power management methodologies for the efficient implementation of low-power dataflow-based reconfigurable designs, the literature about the power management techniques will be deeply analysed in the technical chapters. In particular Chapter 4 and Chapter 5 analyse the automated methodologies for the implementation of power saving strategies in ASIC systems, Chapter 6 focuses on the power management techniques in dataflow field, while Chapter 7 illustrates the effort made in the FPGA field, to develop mixed-grain architectures as a way to offer different tradeoffs between performance and power consumption.

2.1 Reconfigurable Computing

Reconfigurable computing refers to a class of digital electronic system architectures that combine the flexibility typical of software programmed systems to the high performance of the hardware implementations. They are usually conceived as collections of processing elements (PEs) whose functionality and connections can be configured at run time, to adapt them to different applications or operating modes [25], a reconfigurable interconnect and a flexible interface to connect the fabric with the external world. Reconfigurable systems are

often called adaptive, meaning that the logic units and interconnects of the system can be modelled to fit a specific functionality by programming it at hardware level [116]. However, the more these components are able to fit the applications requirements, the slower they are with respect to less flexible component, that can easily turn out to be also smaller in area and less power consuming [117]. The most common example of reconfigurable platforms are Field Programmable Gate Arrays (FPGAs), which typically provide a very high degree of flexibility through a large density logic substrate.

Such systems can be classified according to the granularity of the PEs and interconnects. The granularity determines the level of detail that each PE can manage and it is typically classified onto fine-grain and coarse-grain. Granularity heavily affects the system configuration process and flexibility. Fine-grain reconfigurable (FGR) approaches involve bit-level FUs, resulting in a high flexibility but requiring long configuration time (due to the amount of logic to be configured within the substrate). Coarse-grain reconfigurable (CGR) systems provides word-level PEs, thus providing less flexibility, fitting better around the wanted functionality, while guaranteeing faster configuration phases.

2.1.1 Fine-Grain Reconfiguration

FGR systems involve PEs that perform simple functions on a single or a small number of bits. Similarly, their interconnects can differentiate the data routing at the same bit level than the PEs one. The most common PEs in FGR systems are the small Look-Up Tables (LUTs) adopted to implement combinatorial logic on commercial FPGAs. These circuits are able to perform any logic function for the supported number of inputs, once configured with a proper bitstream. FGR systems involve a huge amount of PEs in order to implement complex functionalities. In turn, the amount of data necessary to configure these PEs can easily become very big and the configuration phase, that is the downloading of the configuration data, gets slow. For this reason the runtime reconfiguration of the FGR devices is hardly achieved.

The most common FGR devices are FPGAs. The latest top range commercial FPGAs, such as the Xilinx Ultrascale+ Family [124]can embed more than one million LUTs with up to eight input each. LUTs in FPGAs are commonly packed in groups along with some bit-wise memory elements (Flip-Flops (FFs)). The blocks involving LUTs and FFs are the effective PEs of FPGAs, by the point of view of the reconfigurable architecture. Thanks to their strong flexibility, FPGAs can be adopted on the practice in different ways [113]: as reconfigurable glue logic for high performance interfacing layers form data sources to processors [19], as hardware accelerators for a single functionality supporting high speed data exchange [38] or as flexible accelerators able to speed-up several software applications within a wider server structure. The strong efficiency, both in terms of performance and power, of these kind of devices is very close to the one typical of application specific integrated circuits (ASICs), especially if compared to general purpose systems like CPUs and GPUs [5].

However FPGAs configuration requires a huge amount of information and time. For this reason the reconfigurability mainly provided by this kind of devices is at design-time. Since the last ten years, FPGAs are able to support also a sort of runtime reconfiguration through the so called partial reconfiguration [122] [6], also known as dynamic partial reconfiguration. Partial reconfiguration is based on the configuration of a portion of the design during the execution by exploiting a set of previously generated configurations stored on a dedicated memory that is accessed from a configuration module. Kohn [55] exploited the par-

tial reconfigurability in order to develop a hardware accelerator for Sobel and Sepia filtering within a full high definition video pipeline. In this case partial reconfiguration allowed the reduction of the configuration bitstream size from more than 4 MB to less than 135 kB, leading to a saving time over 95% with respect to a full system reconfiguration.

Besides FPGAs, other kinds of FGR systems have been proposed in literature. Chiu et al. [51] developed a FGR architecture whose PEs involve two four-ports LUTs and one FF, similarly to the common FPGA ones. The architecture, called FMRPU, is organised as an array of 4096 overall PEs with a hierarchical interconnect layer that guarantees short combinatorial paths. FMRPU has been conceived for high throughput and data parallel applications. It has been validated and compared with some similar devices on motion estimation and digital signal processing operations. Other solutions, like the one proposed by Agarwal et al. [4], differentiated the PEs by packing inside only combinatorial logic: four 3-input LUTs and three 4-bit ripple-carry adders. The PEs can be configured (43 configuration bits are required for each PE) in order to implement generic logic functions with up to five inputs, one 4-bit adder in parallel with an 8-bit one, one 4-input 4-bit adder or a 4-bit multiplier. The FFs needed to store operands and intermediate results are integrated in the interconnect layer as a register bank with 64 entries 32-bit each. A unique register bank is provided to serve six different PEs. The architecture, prototyped on a 32 nm CMOS technology, is intended to provide scalable and high efficiency computing power on microprocessor platforms for the digital signal and media processing.

2.1.2 Coarse-Grain Reconfiguration

In order to overcome the problem of slow configuration phases on FGR systems, CGR architectures waive some flexibility by adopting bigger PEs. CGR PEs are typically constituted by arithmetic and logic units (ALUs) along with a significant amount of storage. They can still be reconfigurable and their interconnect is typically at the same coarse granularity. CGR architectures can also achieve higher area efficiency and simpler placement and routing phases than FGR ones [41]. In literature several different CGR systems have been proposed. Hartenstein [41] introduced a classification for CGR architectures based on the layout and interconnections of the involved PEs: linear arrays, mesh-based and crossbar-based.

CGR architectures organised as single or multiple linear arrays are very suitable for the mapping of execution pipelines. Within linear arrays, natural interconnects are the ones connecting each PE with the two nearest neighbour ones, the predecessor and the successor in the pipeline. However, in presence of forks, additional links spanning among the whole array and proper routing resources are required. An example of a linear array is given by Smaragdos et al. [108]. Authors proposed a fault tolerant multi-core architecture, where pipeline stages (the CGR architecture PEs) of the cores are interleaved by switching elements. A faulty stage within a core can be replaced by the same stage belonging to a different core. In some cases, linear arrays are used to implement CGR platforms similar to VLIW processors, such as for the Montium processing tile [45]. This reconfigurable processing unit involves five *processing parts* composed by one ALU, two overlying memories and related registers bank and interconnect.

The most common class of CGR architectures is the mesh-based one. It is basically a rectangular two dimensional array of PEs, where the most common interconnects are 4 (North, South, East, West) or 8 (including also the diagonal directions) nearest neighbour short links. Examples of these kind of architectures are the works of Niedermeier et al. [76] and Paul et

al. [83]. Niedermeier et al. [76] propose an eight by eight two dimensional mesh of PEs, performing arithmetic and logical operations on integer or fixed point numbers, with an additional first column of memory elements. Paul et al. [83] proposed reMORPH, a target specific platform (built upon a Xilinx FPGA) that exploits DSP48 and BRAM coarse-grain blocks of the FPGA as cores of the array PEs.

The last type of CGR architectures introduced by Hartenstein [41] is the crossbar-based, that relies on a crossbar switch, the most powerful communication network. Despite its adoption in FGR systems is very common, a full crossbar is only rarely employed in CGR architectures. The crossbar is rather used in different reduced configurations. One of the first works on crossbar-based CGR architectures is PADDI-2 [127]. It adopts a two level communication structure composed by a local network, connecting four PEs (16-bit ALUs) within a unique cluster, and a global crossbar network, connecting the various clusters. Inoue et al. [50] explored the trade-off of having different crossbar configurations within a system involving the same VGLC as [1]. The authors showed that, by reducing the routing tracks of a full crossbar solution, the area and the same routing performance can be improved. The cost that has to be paid for this improvement is a decreasing of the processing speed in terms of maximum operating frequency.

Modern trends of embedded devices that have to deal with several and strong constraints are pushing the CGR systems to non conventional architectures. In particular, the above mentioned approach inversion that occurs going from general purpose systems to ASICs may affect also the CGR systems design. The applications that have to be mapped within the CGR platform are not longer fitted within the prefixed architecture, eventually exploiting the provided configurability. Rather, the CGR system is shaped exactly around the wanted set of functionalities, maximising the efficiency in terms of power, resources usage and performance. Obviously, the flexibility versus efficiency trade-off still applies and the price paid with this strong specialisation is an equally strong decrease of flexibility. At any rate, extremely application specific CGR systems lead to non conventional architectures that are not classifiable to any of the categories proposed by Hartenstein [41]. As it will be clearer in the following chapters, this thesis work focuses on such extremely application specific CGR systems.

Other works explored the possibility of taking advantage by adopting different kinds of granularity on the same substrate, in order to achieve both high flexibility guaranteed by FGR architectures and strong performance obtained by CGR ones. Modern FPGAs themselves are actually multi-grain systems. Indeed, even though they are substantially FGR architectures, provide some CGR functional blocks and memories. Indeed, when adopted to implement arithmetic functions, fine-grain structures require more area, latency and consume more power than the corresponding coarse-grain blocks [117]. A deeper overview of multi-grain systems is given in Chapter 7

2.1.3 Composition

Depending on the kind of the involved PEs, reconfigurable architectures can have different compositions. In particular it is easy to distinguish two main kinds of systems: homogeneous architectures, that involve a unique kind of PEs, and heterogeneous architectures, that exploit several kinds of PEs. Homogeneity is very common in reconfigurable architectures, since it eases the process of mapping the applications within the PEs. Among the architecture examples previously presented there is plenty of homogeneous architectures either

for FGR ([4] [51] and CGR ([45] [76] [83]) systems. Homogeneity is also somehow related to the flexibility versus efficiency trade-off. On the one hand, the availability of identical and replicated PEs limits the degree of specialisation, and in turn efficiency, that the architecture configurations can achieve if compared with dedicated heterogeneous blocks. On the other hand, homogeneous PEs are more generic than heterogeneous ones and can implement a wide number of functionalities.

The most common homogeneous PEs are ALUs. They generally perform arithmetic (sum, subtract) and logical operations (shift, compare) on two or more integer and in some cases fixed point numbers. Sometimes also the multiplication is supported, but it is also often performed by dedicated nodes in heterogeneous architectures. Depending on the kind of addressed applications, the ALUs can also support floating point arithmetic. PEs often embed some memory resources, in terms of input/output registers bank or as a small local memory.

As the requirements of efficiency for applications execution has become stronger, the diffusion of heterogeneous architectures increased. FPGAs started integrating dedicated memory and multiplier blocks on their texture in order to avoid the waste of resources needed to implement these functionalities on generic logic. The introduction of these additional elements within the array boosted the performance for some applications, such as digital signal processing ones. Newest FPGA devices integrate also more complex blocks, like digital signal processors, able to perform a variety of multiply and accumulate functions.

The border between the two composition classes, homogeneity and heterogeneity, is very thin and strongly depends on the point of view of the observer. The architecture proposed by Niedermeier et al. [76] involves a two dimensional array of identical computational PEs, plus an additional first column of memory resources. Thus this architecture can be seen as an heterogeneous system. The Montium processor tile [45] introduced before and so far considered as homogeneous architecture is, actually, an heterogeneous structure if the memory blocks are considered as PEs.

2.2 System Level Modelling: The Dataflow Paradigm

Model-based design has been widely studied and applied over the years in many domains of embedded processing. Dataflow is well-known as a paradigm for model-based design that is effective for embedded digital signal processing (DSP) systems [12]. A dataflow can be described as a direct graph $DFG\langle V, E\rangle$, where V is the set of vertices of the graph (the actors) and E is the set of edges representing loss-less, order-preserving point-to-point connection channels. The dataflow paradigm has been firstly proposed in the early 1970s with the works of Dennis [31] and Kahn [40]. In particular, Kahn [40] introduced a distributed model of computation called Kahn Process Network (KPN). In KPN processes communicate by means of unbounded unidirectional First-In-First-Out (FIFO) channels. Each process is executed sequentially, while the overall computing at the processes level is parallel. A special case of KPNs are the Dataflow Process Networks (DPNs), firstly proposed by Lee et al. [59]. DPNs describe programs through the interaction of logical entities, the *actors*, analogue to the KPN processes but providing conditions on which an actor is ready to execute (*firing rules*). Figure 2.2 illustrates an example of DPN. The actors are abstract representations of PEs that encapsulate their own internal state and generate output tokens from the respective inputs, asynchronously concurring to the whole computation. The communication between actors is based on the exchange of sequences of atomic data packets called *tokens*. This commu-

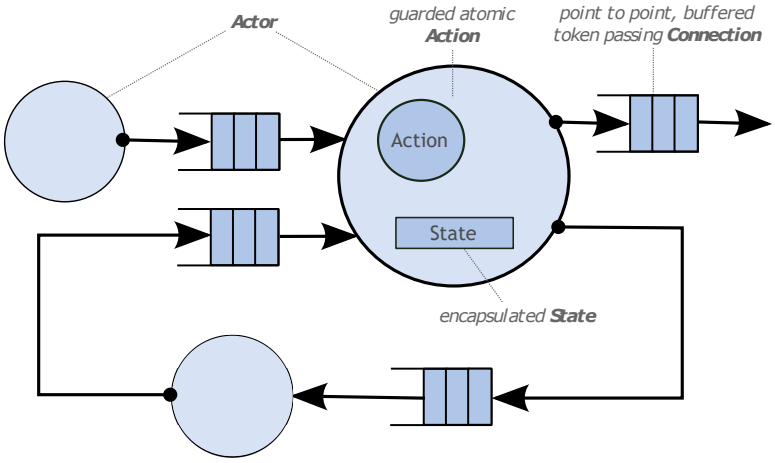


Figure 2.2: Dataflow Process Network design example.

nication is asynchronous, since it is driven by the production and consumption of tokens. Once triggered for processing (*fired*), actors execute a sequence of steps called *actions* that can result in:

- the consumption of one or more input tokens;
- the production of one or more output tokens;
- the change of the actor internal state.

Actors are implemented by any host language able to specify the actions firing rules. Modularity lets it possible to combine and make communicate actors described through different specification languages such as Intellectual Properties (IPs) coded in HDL, low-level software actors written in C and high level software actors written in Java. Actors may be atomic actors or sub-graphs encapsulating in turn a dataflow network in a hierarchical fashion. Such a kind of model is very suitable to manage the concurrency due to parallelism that one application may intrinsically have. Indeed, thanks to the token mediated communication policy, race conditions among actors are avoided. Furthermore, dataflows are highly modular specifications naturally amenable to block diagrams, therefore perfectly fitting to signal processing applications. Modularity strongly favours the code reuse, speeding-up the time to market needed for modelling updated versions of existent applications or new functionalities by scratch. For these distinctive features, the dataflow paradigm constitutes a valid alternative to the standard imperative programming model when concurrency occurs (e.g. in multi-core platforms). The imperative paradigm implements concurrent programs through threads, but leaves the onerous task of managing the concurrency among them to the programmer. In a dataflow program, concurrency depends on the token availability and each actor may fire independently without taking care of the other actors execution state. In this way, the program can be distributed over different PEs and the application parallelism is easily exploited. All these distinctive features make dataflows very suitable for programming highly parallel, also heterogeneous systems, like multi-processor SoCs or CG reconfigurable arrays.

2.2.1 Dataflow Models of Computation

In literature, several dataflow models, also referred as dataflow Models of Computation (MoCs), have been proposed. The differences between dataflow MoCs are related to the actors internal communication and actions scheduling. In particular the number of exchanged tokens can be fixed, variable, not specified or it can depend on parameters. Furthermore actors can have an external control flow.

The *Synchronous DataFlow networks* (SDF) [58] is a special case of DPNs where actors have static firing rules: the number of tokens produced and consumed by each action is fixed. SDF MoC makes it possible to determine if the program can be scheduled at compile-time by means of a static code analysis. The static (compile-time) analysis is based on the checking of consistency and schedulability, that is the ability to come back to the initial FIFO states. The former is checked through the extraction and resolution of the topology equation, by the rank of the related topology matrix. The latter is checked by verifying that enough initial tokens have been set. The static firing analysis produces, if the SDF is consistent and schedulable, a static schedule (a predefined sequence of actor firings) with a bounded memory usage and without deadlocks.

For many signal processing applications, it is not possible to represent all of the functionality in terms of purely decidable dataflow representations. For example, functionality that involves conditional execution of dataflow subsystems or actors with dynamically varying production and consumption rates generally cannot be expressed in decidable dataflow models [12]. Dynamic dataflow models are dataflow modelling techniques, most suitable for DPS systems, in which the production and consumption rates of actors can vary in ways that are not entirely predictable at compile time [12]. Most existing DSP-oriented dynamic dataflow modelling techniques do not provide decidable dataflow modelling capabilities. The increased modelling flexibility (expressive power) provided by such techniques is obtained by giving up guarantees on compile-time buffer underflow (deadlock) and overflow validation. Analysis techniques may succeed in guaranteeing avoidance of buffer underflow and overflow for a significant subset of specifications, but specifications may arise that "break" these analysis techniques. Several dynamic dataflow techniques involve different kinds of modelling abstraction, suitable for DPS systems.

The *Boolean dataflow* (BDF) model of computation extends SDF with a class of dynamic dataflow actors in which production and consumption rates on actor ports can vary as two-valued functions of control tokens, which are consumed from or produced onto designated control ports of dynamic dataflow actors. An actor input port is referred to as a conditional input port if its consumption rate can vary in such a way, and similarly an output port with a dynamically varying production rate under this model is referred to as a conditional output port. It has been proven that BDF is Turing complete, and also that if a given model of computation can express all SDF actors as well as the functionality associated with the BDF switch and select actors, then such a model can be shown to be Turing complete [14].

The MPEG RVC standards ISO/IEC 23001-4 and 23002-4 adopt a DPN MoC (more precisely on a DPN variant that allows token peek) for modelling the applications. DPN has been adopted in video coding, as well as in other different multimedia areas, due to its high expressiveness and to the availability of a formal programming language, the *Caltrop Actor Language* (CAL) [35], supporting all its features. CAL is a C-like textual programming language developed in order to be easy to use through the adoption of a minimal semantic core, and directly captures the description of DPN actors. Connections are implemented

as FIFO channels, which are used to transmit tokens.Specific token configurations or sequences can fire actions inside the actors, according to predetermined guard expressions. Guards are boolean expressions on the current state and/or on input sequences that need to be satisfied for enabling the execution of an action. The actions execution may lead to state variations or to specific output tokens emission. For the specification of the interconnections among the DPN actors, RVC adopted an XML dialect called XML Dataflow Format (XDF), that is part of the related standards too.

The *CAPH programming language* incarnates the dataflow model by formalizing the behaviour of each dataflow actor and the description of the kind of values exchanged between actors and of the computations that can be performed on these values. Similarly to CAL, CAPH is suitable for stream-based applications. However, CAL provides constructs for expressive guards, priorities and scheduling, while scheduling in CAPH is much simpler, being specified by a pattern-matching rule-based mechanism. Also CAL uses a dedicated network language (XML) while CAPH exploit functional expression and higher-order constructs.

Parameterized Synchronous DataFlow (PSDF) is a meta-modelling framework for integrating dynamic parameters into the class of dataflow models of computations that that present graph iterations like SDF [11]. This allows actor tokens production/consumption rates to be parameteric, and in turn dynamically reconfigurable. In the class of PSDF, πSDF [32] introduces also interface based hierarchy [86] and dependences among different parameters in order to reconfigure the production and consumption token rates of actors at runtime.

Enable-invoke dataflow (EIDF) is a general dataflow model of computation that supports the dynamic behaviour of the dataflow actors [88]. In EIDF, each actor is divided into a set of *modes*, where each mode, when executes, consumes and produces a fixed number of tokens, i.e., has a fixed *consumption rate* and *production rate* associated with each input and output port, respectively. The dynamic behaviour can be achieved by switching among these modes at runtime. Each mode is specified by an *enable method*, which checks whether there is sufficient data available on the actor's input ports to fire the actor in its current mode, and an *invoke* method, which executes the current mode, consumes and produces data with the fixed rates of current mode, and returns a set of admissible *next modes*. Any mode in the set of next mode could be further checked for readiness by the enable method and then invoked, thus the non-deterministic applications could be modelled by EIDF.

Core functional dataflow (CFDF), which the LWDF programming approach is based on, is a special case of the EIDF model. Instead of returning a set of valid next modes for the next actor invocation, the invoke method should only return one valid mode of execution, which means the actor execution could only proceed down one deterministic path. Thus, CFDF is a restricted version of EIDF so that the returned set of the next modes only contains a single element. This characteristic ensures a deterministic application.

The most important feature of EIDF and CFDF is a clean separation of enable and invoke capabilities. Once an invoke function is executed, it assumes that sufficient data is presented, since the firing condition has already been checked by its enable function. This feature improves the predictability of an actor firing and facilitates efficient scheduling techniques. Such design imposes restrictions only on the structure and interface of the dataflow actors instead of the actor functionality, which enables the retargetability of this dataflow modelling approach.

2.2.2 DSP-oriented Dataflow-based Tools

Due to their distinctive features, dataflow MoCs are adopted in a wide variety of tools for both software and hardware design. In [70], methodologies for modelling, implementing and optimizing pipelined hardware component networks from a high level dataflow graph description are developed. They offer the possibility of optimizing the design in terms of throughput or resource consumption. Stefanov et al. [110] present a system design flow, centred around exploiting the Kahn Process Network model, in which an application written in a subset of Matlab is mapped onto a target platform composed of a CPU and an FPGA in a systematic and automated way. In realizing the flow, they developed and used the COMPAAN and LAURA tools, that allow us to go from an application specification in Matlab to an implementation of the application running on the target platform. PREESM is an open-source Eclipse-based framework that provides dataflow-based methods to study a multicore DSP system [85]. PREESM provides the designer with information on algorithm parallelism and latency, as well as on system memory requirements. It automatically maps and schedules the application, specified as πSDF MoC, over the available PEs, and provides a code generation to transform the dataflow representation into a compilable code.

Around the MPEG RVC and the adopted CAL programming language, a wide variety of design tools have been developed in literature. Most of them leverage on the Open RVC-CAL Compiler (ORCC) [24], a compilation infrastructure in charge of generating descriptions in several languages (software, hardware or mixed for co-design [107]) starting from RVC-CAL actors and XDF networks. At the moment ORCC is provided as an Eclipse plugin written in Java and relies on an Intermediate Representation (IR) of the DPNs that is still specified in Java. The IR can be exploited to feed several other tools such as Turnus [21], which offers simulation, profiling and design space exploration capabilities, and Xronos [10], in charge of providing HLS for Xilinx FPGAs. The ORCC compilation infrastructure, as well as the MPEG-RVC framework itself, is continuously evolving in order to support more and more advanced features and to produce more and more dynamic systems. The Multi-Dataflow Composer (MDC) tool is a framework for the automatic creation of multi-functional reconfigurable platforms, that performs a complete design space exploration, evaluating the trade-off among resource usage, power consumption and operating frequency [82]. In [75], Nezan et al. presented the integration of ORCC with the MDC tool, that automatically optimize dataflow specifications to generate coarse-grain reconfigurable HDL designs.

Synflow Studio [96] provides a unified HLS framework leveraging on a C-based high level proprietary dataflow language named Cx. Cx can be compiled using Synflow Studio into Verilog or VHDL code that is compliant with a variety of design tools, including simulation, timing analysis, test analysis, and synthesis tools. The CAPH language and framework represent another recent effort to generate HDL from a dataflow language [99, 100]. More precisely CAPH is a toolchain built around the domain-specific language for the specification of stream-processing applications based on a dynamic dataflow MoC. This latter is specified through a functional language named Functional Graph Notation (FGN) [98], allowing a complete description of a dataflow network by means of purely functional expressions, and resulting in improved abstraction capabilities, easier wiring description and more efficient errors check. The Lightweight dataflow (LWDF) is a programming methodology that allows designers to systematically integrate and experiment with dataflow modelling approaches in the context of existing design processes [103]. LWDF is “lightweight” in the sense that the programming model is designed to be minimally intrusive on existing design methodologies

and processes. It provides providing a compact set of APIs that can be used to incorporate advanced dataflow techniques and requires minimal dependence on specialized tools or libraries.

2.3 Power Issue in Digital Circuits

Nowadays small portable devices are required to efficiently execute multiple fancy functions. The huge diffusion of portability in embedded devices has strongly raised the importance of the energy consumption (given as $AveragePowerConsumption \times ExecutionTime$) and, in turn, the power consumption constraints for the designers due to the battery life limits. The power issue has become so critical that it has been defined the "new timing constraint", since timing has historically been the most important constraint for the embedded system development. However the raising of power constraint, especially within specific application fields (e.g. biomedical image processing), does not relax the other main design requirements related to area and performance (that is tightly coupled with timing). In general, area minimisation leads also to power minimisation, so that meeting area and power constraints together may not be very hard. On the contrary, the performance maximisation is typically in contrast with the requirement of limiting power consumption and the realisation of a power aware execution efficient system could turn out in a real nightmare.

In digital systems, power consumption can be divided onto two main contributions: static and dynamic (see Equation 2.1). The former is always present since it is due to leakage currents (P_{lkg}). The latter is dissipated only when logic transitions occur, so that it is related to the switching activity during the system execution.

$$P_{tot} = P_{static} + P_{dynamic} \tag{2.1}$$

Dynamic power has always been several order of magnitude larger than the static one, so that designers focussed their power saving effort mainly on the dynamic contribution. However, in the last decade the transistor size is getting smaller, causing the static power to increase its weight in the power equation due to its own growth and to the contextual decrease of the dynamic power. With technologies below 90 nm, designers are required to minimize both static (P_{static}) and dynamic ($P_{dynamic}$) terms.

The static contribution of the power consumption has only recently started to being strongly taken into consideration by the designers. This dissipated power is only due to the fact that the system is powered on. As previously said, the dissipation occurs since, within a CMOS device, transistors present non ideal leakage currents between different areas of the CMOS transistors. The static power can be quantified as the product between the overall leakage current of the device, $I_{leakage}$, and the supply voltage, V_{supply}.

$$P_{static} = I_{leakage} * V_{supply} \tag{2.2}$$

The overall leakage currents involves following main contributes:

- sub-threshold leakage: sub-threshold currents flowing from the drain to the source of a transistor operating in the weak inversion region.

- gate leakage: tunnelling current which flows directly throughout the gate insulator (especially for very thick channels).
- reverse-bias junction leakage: current between diffusion regions, wells and substrate, caused by minority carrier drift and generation of electron/hole pairs in the depletion regions.

The main approaches that aim at reducing static power consumption act mainly on the supply voltage V_{supply} and on the transistors threshold voltage V_{th}. The reduction of the supply voltage clearly leads to a reduction of the whole static dissipation. In particular the main approaches acting on V_{supply} are the multi-supply voltage, that partitions the device in different areas driven by different supply-voltages, and the power-shut off or power gating, that avoid unnecessary static consumption by shutting off the logic when unused. Note that, despite being beneficial under the power aspect, reducing the supply voltage makes transitions slower: at this purpose multi-supply voltage approaches typically keep higher the voltage of the logic involved on the system critical path.

While the impact of the V_{supply} in reducing static power is straightforward, since it is directly involved in Equation 2.2, the effects of acting on V_{th} are not so immediate. Generally speaking the reduction of V_{th} can be beneficial to the system performance, since transistors earn speed during their active state. However, static power has an inverse proportionality with respect to the threshold voltage: in terms of consumption, higher V_{th} are generally beneficial. As for the supply voltage, a common solution is to have different threshold voltages for different logic areas, basing on their criticality in terms of timing. The implementation of power awareness systems that involve static consumption limiting techniques is usually not trivial. As a matter of fact, static saving approaches require strong resources overhead and dedicated support from the target technology library. For instance power gating employs three different kinds of dedicated cells (sleep transistors, isolation and state retention cells) that can be instantiated several times, depending on the design.

The dynamic contribution of the power dissipation exclusively occurs when system nodes switch their state. It can be divided also as:

$$P_{dynamic} = P_{trans} + P_{cap} \tag{2.3}$$

where P_{trans} is the transient power consumption, due to the short cut currents between power supply and ground when the gate is changing its state. P_{cap} is the dissipation caused by the charging and discharging of the parasitic capacitances during logic transitions. Both P_{trans} and P_{cap} are directly proportional to the switching activity (total number of output switching per gate), to the clock frequency and to the square of the voltage supply V_{supply}. All the techniques acting on this latter quantity, previously presented as beneficial for static power, can bring a positive impact also on the dynamic contribution. Anyway, the most approaches aiming at reducing the dynamic power act on the system clock. In particular the main trends are related to shutting off the clock signal (clock gating) at all or to provide multi-frequency environment. The multi-frequency approaches push at the maximum frequency only the most critical resources (the resources with the longest execution latency) within the system, while they slow down the not critical ones. This strategy may lead to a solution with the same performance of the design where all the logic is driven by the same maximum frequency, but with limited consumption. Further techniques and tools for power management

and minimization of average power consumption in digital systems will be deeply analysed in the literature of the technical chapters.

Chapter 3

MDC: Multi-Dataflow Composer Tool

This chapter describes the Multi-Dataflow Composer (MDC), an automated framework for the generation and management of coarse-grain reconfigurable (CGR) multi-functional architectures. First version of MDC was developed at the University of Cagliari, and has been exploited in the work presented in this thesis as primary tool for the exploration and deployment of power management techniques in dataflow-based CGR systems. Indeed, MDC is meant to address the difficulty of mapping a set of different applications onto a CGR architecture [20, 57], combining together a set of input dataflow specifications that describe desired system behaviours. It shares dataflow actors through a datapath-merging problem-solving algorithm and generates a CGR hardware substrate [80]. MDC is composed of four main components:

- *Baseline MDC Core*: performing dataflow-to-hardware composition, by means of datapath merging techniques.
- *Structural Profiler*: performing the design space exploration of the implementable multi-functional systems, which can be derived from the input dataflow specifications set, to determine the optimal CGR substrate according to the given input constraints.
- *Dynamic Power Manager*: performing, at the dataflow level, the logic partitioning of the substrate to implement at the hardware level a clock gating strategy, reducing the dynamic power consumption.
- *Coprocessor Generator*: performing the complete dataflow-to-hardware customization of a Xilinx compliant multi-functional accelerator. Starting from the input dataflow specifications set, such an accelerator can be either loosely coupled or tightly coupled, according to the design needs, and also its drivers are derived.

Figure 3.1 illustrates the development timeline of MDC tool and its components, until the end of 2015, period in which this research thesis started. Every extension of MDC, related to the work here presented, is described in the technical chapters of this thesis. Following section are meant to provide a deep explanation of the MDC Tool components as they were at the end of 2015.

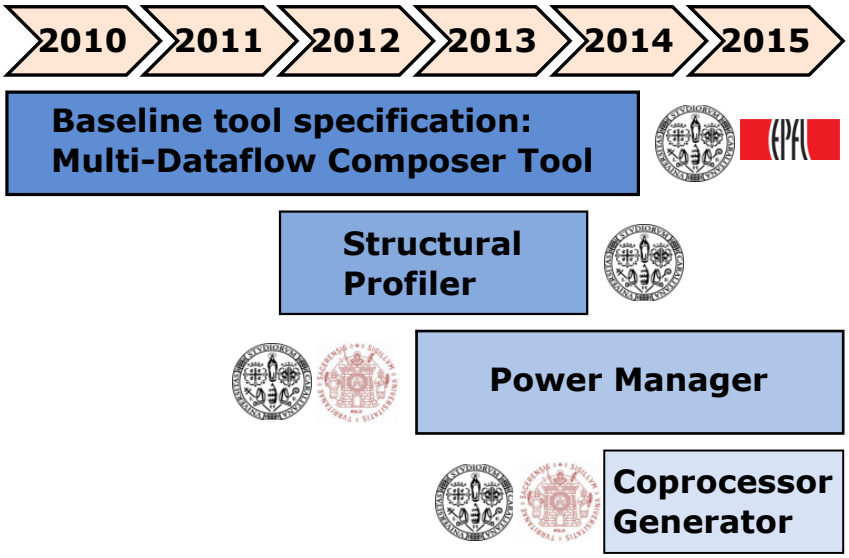


Figure 3.1: Multi-Dataflow Composer Tool - Development Timeline.

3.1 Baseline MDC Core

The core functionality of MDC tool is in charge of mapping a set of dataflow specification onto a CGR substrate, automating the mapping process while minimizing hardware resources [80]. This issue is known in literature as the datapath merging problem. MDC solves it by exploiting two different solutions: (1) a heuristic algorithm [80], or (2) the application of the moreano algorithm [71].

The tool is designed to be connected to higher-level utilities by means of an adequate front-end, in charge of parsing the high-level descriptions of the datapaths to be combined. In this way, relying on the chosen front-end, MDC is able to process any type of DFGMDC has been couples with different dataflow-based tools, such as ORCC [24], CAPH [99] and Synflow [96] (see Section 2.2.2). In this Chapter, the coupling between ORCC and MDC, and the DPNs (expressed as XDF files) are used to illustrate MDC features.

Figure 3.2 shows an overview of the coupled ORCC-MDC design flow. Three major steps are required to generate the HDL description of a multi-functional reconfigurable architecture, starting from the DPN models of the functionalities to be implemented:

- Input DPNs parsing.
- Multi-dataflow generation.
- CGR hardware architecture generation.

ORCC parses the input DPNs, along with their actors, and translates each of them into a DFG Intermediate Representation (IR). During the parsing ORCC explodes non-atomic actors (composed of a sub-network of actors), flattening the input DPNs. Then the MDC front-end leverages on such IRs to assemble a single multi-functional specification (*multi-flow IR* in Figure 3.2). MDC front-end also keeps trace of the system programmability through the *Configuration Table* (*C_TAB* in Figure 3.2). Reconfiguration is implemented by multiplexing resources in time. Ad-hoc low overhead switching modules (Switching Boxes - SBoxes) are

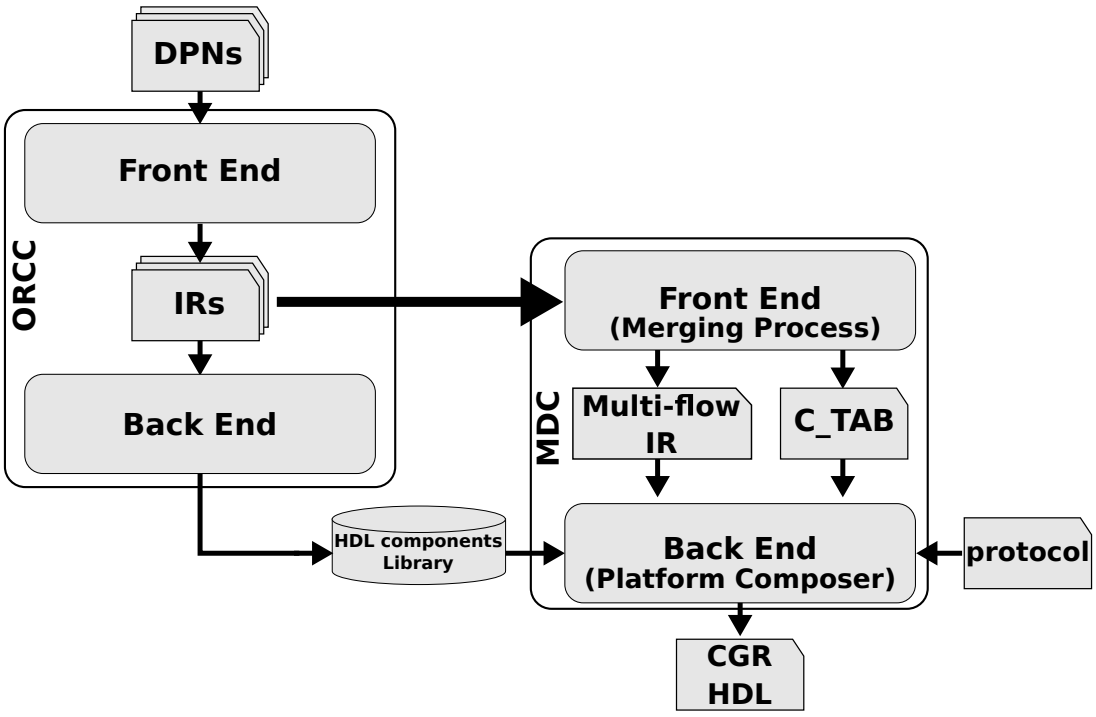


Figure 3.2: Multi-Dataflow Composer tool: an overview.

placed at the crossroads between the different paths of data and driven by dedicated Look-Up Tables (LUTs), whose content is defined according to the *Configuration Table*. Once the input DPNs have been merged, the MDC back-end creates the HDL CGR hardware (*CGR HDL* in Figure 3.2), mapping each actor onto a different FU. Even though MDC is coupled with ORCC, the generated CGR hardware is not restricted to the RVC-CAL communication protocol. Indeed, MDC takes as input an XML file that describes communication protocol between FUs. Thus, MDC is actually able of considering a dataflow network as generic graph, where communication among FUs can be managed with or without First-In First-Out (FIFO) connections, and where the FUs can even be purely combinatorial. The HDL description of the FUs are passed as input to MDC, together with any other necessary module (FIFOs, Fanouts, Memories) within the *HDL components library* that can be manually written or automatically created by HLS tools. In the tool flow shown in Figure 3.2, the *HDL component library* is created by an ORCC backend.

In the current HDL implementation, SBoxes are combinatorial multiplexers; therefore, no dedicated FIFO buffers are inserted for the SBox units. Nevertheless, the FIFOs of the upstream/downstream actors have to be managed. $Sbox\_1x2$ units, inserted to split a path of data, require one FIFO for each outgoing connection. In the case of $Sbox\_2x1$ units, inserted to access a common shared actor, the FIFO buffers are placed before the SBox along the incoming connections. Since the SBoxes are fully combinatorial and the FIFO buffers always belong to the other actors, the well known dataflow problem of the FIFO buffers optimal sizing does not affect the MDC merging process. Input DPNs have only to be properly sized before the MDC execution.

Figure 3.3 shows in detail how the reconfigurable multi-functional architecture is derived. It consider an example with three different DPN specifications (α, β and γ), and the generated output is the HDL description of the CGR. architecture. In this example the MDC

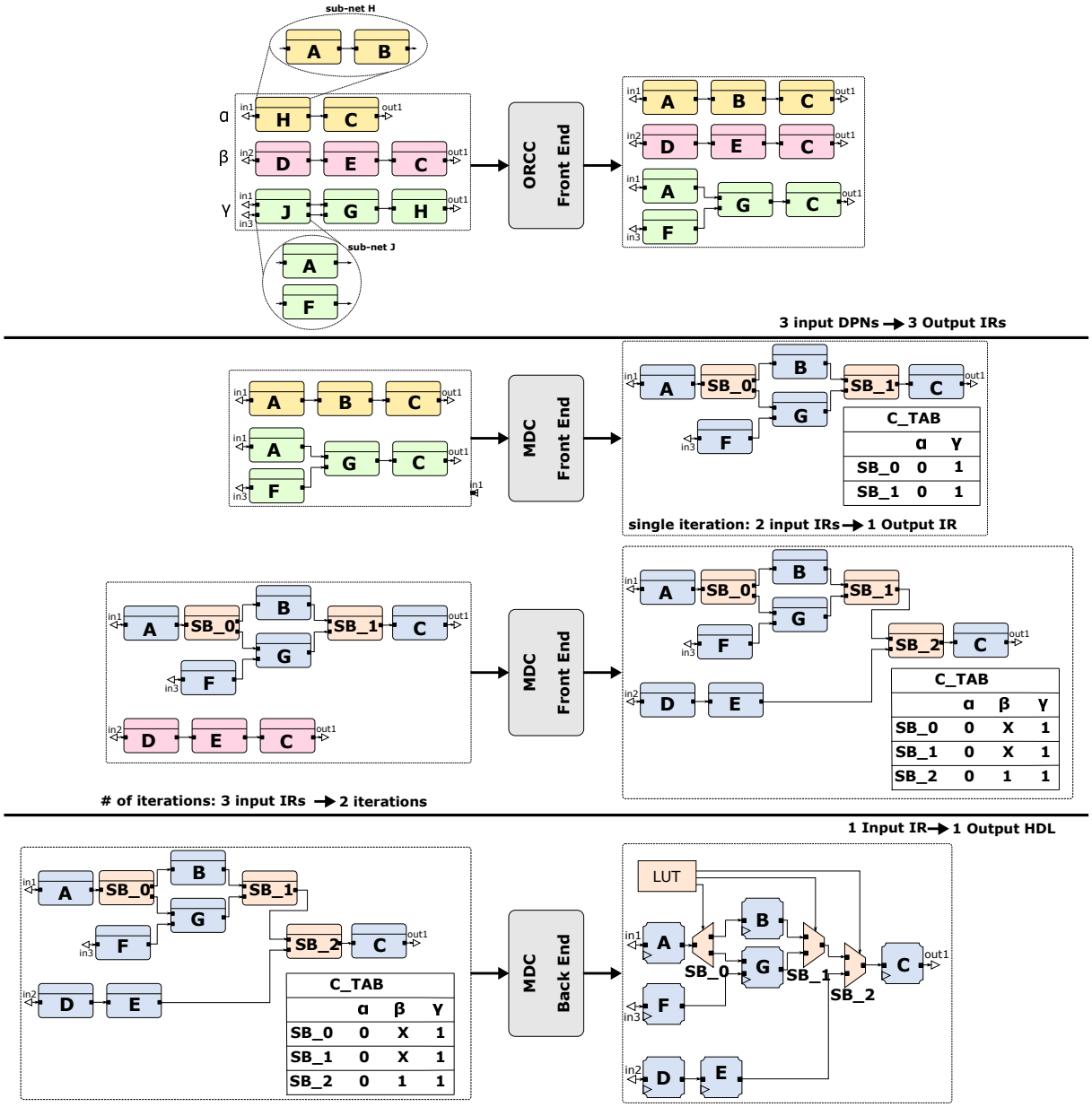


Figure 3.3: Baseline MDC Core: a step-by-step example.

heuristic merging algorithm is considered. At first ORCC parses the input DPNs, flattens the hierarchical actors and builds the corresponding IRs. In the considered example β is already flattened, being composed of atomic actors, while the actor H of α and the actor J of γ enclose a sub-network each. Thus, these latter are exploded in the flattened network.

After parsing the input DPNs, MDC starts the iterative merging process. MDC front-end analyses the IRs in pairs to determine which actors can be shared by the two considered networks. Identical actors are shared in the output IR by introducing dedicated switching elements, used to fork ($Sbox\_1x2$) or re-join ($Sbox\_2x1$) the path of data. It is important to notice that for N input IRs, N-1 iterations are required to complete the merging process and, in the worst case scenario the process can end up with N-1 cascaded SBoxes to access a FU shared by all the N input DPNs. In the considered case with only three input networks, two iterations are required. In the first run, the merging algorithm identifies actors A and C as

identical among α and γ, so it inserts two SBoxes. Then, in the second run, the algorithms identifies actor C as identical among the previous generated *multi-flow IR* and β; thus, only another SBox is inserted. During each iteration MDC assigns an identification value to each network and, for each of them, keeps trace of the right selector values to be assigned to each *SBox* updating the C_TAB.

At last the MDC back-end generates the *CGR HDL*, mapping the different actors of the *multi-flow IR* over the FUs provided within the *HDL components library*. The control signals of the physical SBoxes are generated by the LUTs, whose content depends on the final C_TAB produced by the MDC front-end, that guarantees the computing correctness of each input functionality.

3.2 Structural Profiler

In the adopted iterative merging algorithm, two networks at a time are processed by MDC. Since SBoxes are combinatorial elements, a long chain of SBoxes could implies a change into the critical path, that may negatively affect the operating frequency. Furthermore, an excessive number of switching elements may overcome the benefits of sharing an actor, causing both area and static power loss. Therefore, in some cases it would be more efficient to merge only a subset of the input DPNs. For these reasons, it is fundamental to determine the (sub-)optimal design specification(s) that have to be merged into the CGR architectures.

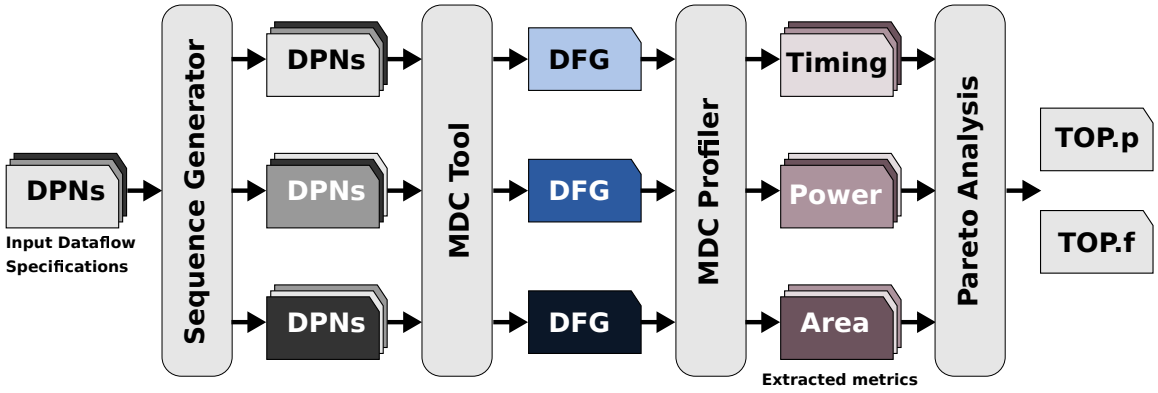


Figure 3.4: Topology Definer: an overview.

The MDC *Topology Definer* is capable of determining the design costs of different implementations, before prototyping, through the back-annotation of low-level information on the different dataflow graphs (DFGs) [81]. Figure 3.4 depicts the implemented profiling-aware topology strategy. The *Sequence Generator* defines all the D possible DPNs sequences that are given in input to MDC according to Equation 3.1, were:

- D_notMer is the *not merged* composition of the N input DPNs in parallel;
- D_Mer is the *all merged* term that, maximizing resource sharing, is given by all the possible permutations of the DPNs merged into a unique one;

- *D_partMer* is the *partially-merged* term that, not following the resource maximization principle, provides all the sequences composed of the combinations[1] that can be extracted from subset of *k* input DPNs placed in parallel with all the permutations of the other *N-k* networks merged together.

$$
\begin{aligned}
&D = D\_notMer + D\_Mer + D\_partMer = \\
&D = 1 + N! + \sum_{k=1}^{N-2} C_{N,k} * (N-k)! \\
&D = 1 + N! + \sum_{k=1}^{N-2} \frac{N!}{(N-k)! * k!} * (N-k)! \\
&D = 1 + N! + \sum_{k=1}^{N-2} \frac{N!}{k!}
\end{aligned}
\tag{3.1}
$$

For each of the different possible DPNs sequences, the MDC tool extracts multi-dataflow DFG as described in Section 3.1. Then the *MDC Profiler* computes the implementation cost for each multi-functional DFG. Computing the implementation cost requires back-annotating the HDL components library with one value of estimated area and power consumption of each DFG. Therefore, given as M the size of the V set, area and power consumption are determined as:

$$Area(DFG) = \sum_{i=1}^{M} a_i \tag{3.2}$$

$$Power(DFG) = \sum_{i=1}^{M} p_i \tag{3.3}$$

Operating frequency can not be estimated simply as a summation as done for area and power. Since only two networks per time are considered during the merging process, different feeding orders may result in different chains of SBoxes that may negatively impact on the critical path (CP). The *MDC Profiler* $\forall n_i \in InN$ (being InN the set of input DPNs) retrieves the corresponding back-annotated CP, CP_i, and defines $CPstatic = max(CP_i)$ as the CP of the non reconfigurable system configuration (with all the given DPNs in parallel).

Then it estimates the longest cascade of SBoxes ($seqSB$) within the considered DFG. Given the number of SBoxes (N_S) that compose the cascade $seqSB$, and given the number of bits of SBoxes data (b), the CP is given by the empirical Equation 3.4.

$$CP\_seqSB = f(b) * ln(N_S) + g(b) \tag{3.4}$$

Coefficients f(b) and g(b) are technology dependent and have been modelled for the target technology interpolating a training set of experimental results, obtained by experiments carried out by means of the RTL Compiler (Cadence SoC Encounter), using ASIC CMOS 90 nm technology as reference, varying the number of SBoxes (from 1 to 100) and the number of bits of SBoxes data (1, 8, 16, 32, 64). Coefficients f(b) and g(b) change depending on the type of SBox (Sbox_1x2 or Sbox_2x1).

[1] A combination is a selection of all or part of a set of objects, regardless to the order in which they are selected. Given A, B and C, the complete list of possible selections of two items would be: AB, AC, and BC.

- Sbox1x2: $f(b) = 0.268 * b + 59.85$, $g(b) = -0.294 * b + 406.3$
- Sbox2x1: $f(b) = 0.114 * b + 87.70$, $g(b) = 0.185 * b + 393.1$

The *MDC Profiler* finally defines $CP(DFG) = max(CPstatic, CP\_seqSB)$ as the maximum of $CPstatic$ and CP_seqSB. In the last step of the *Topology Definer*, a Pareto-based analysis is carried out exhaustively on the entire design space to determine the optimal system configuration(s) according to the selected design effort. For power management purposes it is extremely important to determine the least consuming configuration, but minimizing power consumption not necessarily implies having also the best operating frequency. Therefore, two sub-optimal $DFGs$ are provided as output: the area/power (*TOP.p*) one and the frequency (*TOP.f*) one.

Exhaustive exploration approaches, where all the design points are characterized in terms of objective functions, may lead quickly to search time explosion. According to Eq. 3.1, the design space size grows with the number of input DPNs. Therefore, it is clearly application specific. In [82] it has been demonstrated that the design space dimension grows with $3 * N!$, where N is the number of input DPNs. Nevertheless, as demonstrated in [81], different properties of the design space can be exploited to define a robust and scalable heuristic algorithm performing approximated Pareto analysis:

- *TOP.p* normally is an "all-merged" solution. By construction, the smallest number of actors would provide the smallest area and power consumption according to Eq. 3.2 and Eq. 3.3.
- *TOP.f* may be a "partially-merged" solution, in fact the fewer networks you merge, the smaller is the CP associated to the SBox units chain $seqSB$. Therefore, if the operating frequency is determined by CP_seqSB you can improve it by limiting $seqSB$, placing in parallel to the rest of the design one of the networks contributing to $seqSB$.

3.2.1 Step-by-Step Example

To clarify the different logic phases of the proposed methodology Figure 3.5 depicts a step-by-step example. The starting points of the strategy are five different DPN specifications (α, β, γ, δ and ϵ). The generated outputs of the structural level step are the two multi-dataflow DFGs, *TOP.p* and *TOP.f*. By applying Eq. 3.1, 321 points constitute the design space as possible system specifications. Among these, the *Topology Definer* is capable of identifying the power optimal and the frequency optimal ones. The former, *TOP.p*, is an *all-merged* solution (the optimal feeding order is β, γ, α, ϵ and δ), while the latter, *TOP.f*, is a *partially-merged* one (α and β are kept in parallel while ϵ, γ and δ are merged).

3.3 Dynamic Power Management

As described in Chapter 2.1, in a CGR system all of the logic necessary to compute the different functionalities are instantiated in the substrate and the configurations are enabled by multiplexing resources in time. When a functionality is executed the rest of the design, not involved in the computation, is in an idle state. Thus, CGR systems benefits from the application of dynamic power management strategies. The part of unused resources in a CGR architecture is fixed at design-time. Thus it is possible to divide it into a set of disjointed

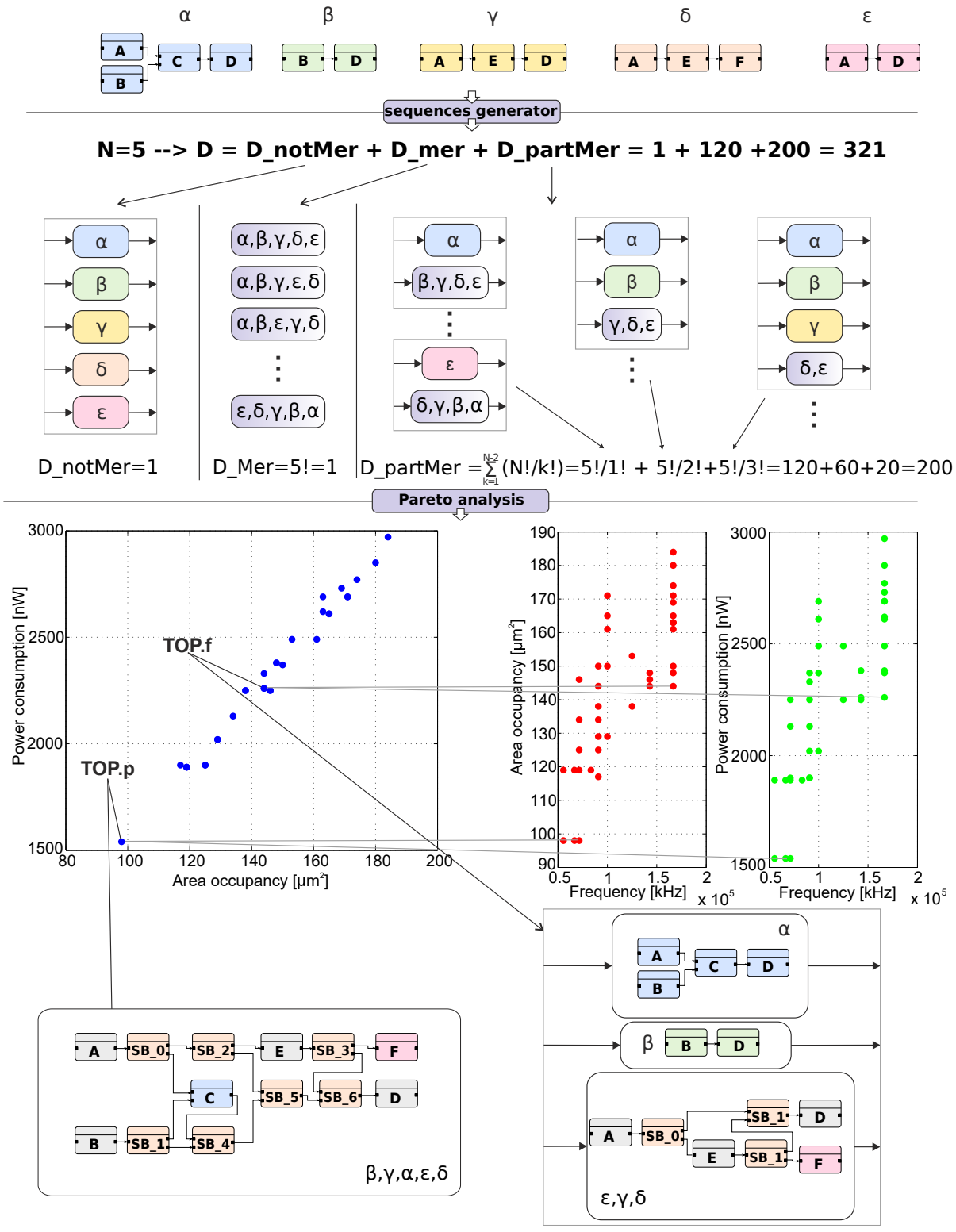


Figure 3.5: Topology Definer: a step-by-step example.

Logic Regions (*LRs*), composed of resources that are always active/inactive together, and reduce their power consumption by applying power saving techniques.

MDC exploits the intrinsic modularity of the dataflow models, to automatically identify the minimum set of *LRs*, by applying the identification Algorithm 1 (see Appendix A) that acts at the specification level. The given dataflows are analysed to identify and group together the actors active/inactive at the same time within homogeneous logic sets and on the MDC GUI users can choose to enable or not the clock gating strategy, to switch off the clock tree of the *LRs* and reduce the dynamic power consumption of the design. The MDC *Power Manager* acts on the (sub-)optimal specification identified by the *Topology Definer* (either *TOP.p* or *TOP.f*) to identify all the *LRs* [82]. It maps each set of actors V_i, belonging the i-th input network, to a set of actors $V_i' \subseteq V$, where $DFG\langle V, E\rangle$ corresponds to the multi-functional dataflow graph.

For each input DPN, MDC extracts its corresponding set V_i' composed of computational actors only. Given S the complete set of *LRs*, MDC minimizes the number of elements, N, within S, in order to minimize the additional logic needed to drive the *LRs*. S contains a partition of the set of actors V_i' of each input network. This means that, when an input network is executed, only its actors are triggered. Beside S, Algorithm 1 defines also the association map (LR_MAP) of correspondences between the input DPNs and the elements of S. At the beginning this map is empty, so that the first V_i' constitutes the first *LR*. For all the other iterations, as shown in the pseudo-code, LR_MAP is not empty anymore and the algorithm extracts, one at a time, all the already identified *LRs* $S = \{S_1, S_2, ...S_P\}$ to be compared with V_i'. Three different situations may occur:

- the current set is equal to an already identified one ($V_i' = S_j$): the association map is updated so that the matching *LR* points also to the current DPN;
- the current set intersects one of the identified ones ($V_i' \cap S_j \neq \emptyset$): a new *LR*, containing the intersected instances, is issued and the pre-determined set S_j and V_i' are modified by removing the intersection. A new entry in the LR_MAP is inserted to match the new set with the current input DPN and all the DPNs already associated to S_j;
- the current set is completely disjointed with respect to the previous ones or there are some elements within it that are not overlapped with any *LR* (at the end of the comparison process ($V_i' \neq \emptyset$)): a new *LR* is created and pushed in S, with the corresponding entry in the association map.

3.3.1 Clock Gating Implementation

MDC exploits the identified *LR* to implement clock gating. It aims to reduce the dynamic power consumption leveraging on the following assumption: the clock of the *LRs* that are not working can be turned off to limit the switching activity of the design and in turn its power dissipation. MDC provides clock gating implementations for either ASIC or FPGA targets. When ASIC target is selected, MDC provides *AND* gates cells (applied directly on the clock to disable it). Otherwise, if FPGA is selected, MDC instantiates for each *LR* to be gated a *BUFG* cell (this can be applied only on Xilinx boards). Targeting FPGA, the number of *BUFG* cells available on the board is limited. If the number of identified *LRs* exceeds the amount of available *BUFG*, MDC provides Algorithm 2 (see Appendix A) to reduce the number of *LRs*, identifying the sub-optimal set of *LRs*, where switching activity in unused FUs is present.

This algorithm merges together two *LRs* at a time, according to one of the following different cost functions:

- minimizing the number of units per *LR* - Given c_i, as the cardinality of the i-th *LR*, and P, as the number of networks activating it, we can define w_{N_i}:

$$w_{N_i} = c_i * P \tag{3.5}$$

  as the *weight* of considered region.

- minimizing the static power consumption per *LR* - Given p_i, as the estimated power consumption (given by Eq. 3.3) of the i-th *LR*, and P, as the number of networks activating it, we can define w_{P_i}:

$$w_{P_i} = p_i * P \tag{3.6}$$

  as the *weight* of considered region.

3.3.2 Step-by-Step Example

Figure 3.6 clarifies the process that leads to identify the minimum set of *LRs*. It considers three input DPNs: α (composed of actors *A*, *B* and *C*), β (*D*, *E* and *C*) and γ (*A*, *F*, *G* and *C*).

1. Firstly the algorithm analyses V'_α, composed with the actors of α.
   - The association map is empty.
   - V'_α is issued as S_1 and a reference from S_1 to α is inserted in the association map.
2. Then V'_β is considered.
   - The association map is not empty anymore and $S = \{S_1\}$.
   - V'_β and S_1 are intersected: they share the actor *C*.
     - The shared actor *C* is issued as S_2.
     - The actor *C* is removed from V'_β and S_1.
     - A reference from S_2 to both α and β is defined.
   - *S* does not have any other element available for comparison.
   - The remaining V'_β is issued as S_3, referencing in the association map just β.
3. Finally V'_γ is considered.
   - The association map is not empty and $S = \{S_1, S_2, S_3\}$.
   - S_1 shares with V'_γ the actor *A*.
     - A new logic region, S_4, containing the actor *A* is issued.
     - A reference from S_4 to α and to γ is pushed in the association map.
     - The actor *A* is removed from S_1 and V'_γ.
   - S_2 is entirely contained within V'_γ.
     - A reference from S_2 to γ is added in the association map.

  - The actor C is removed from V'_γ.
- S_3 and V'_γ are disjointed.
- S does not have any other element available for comparison.
- The remaining V'_γ, containing the actors F and G, is issued as S_5. It matches just γ in the association map.

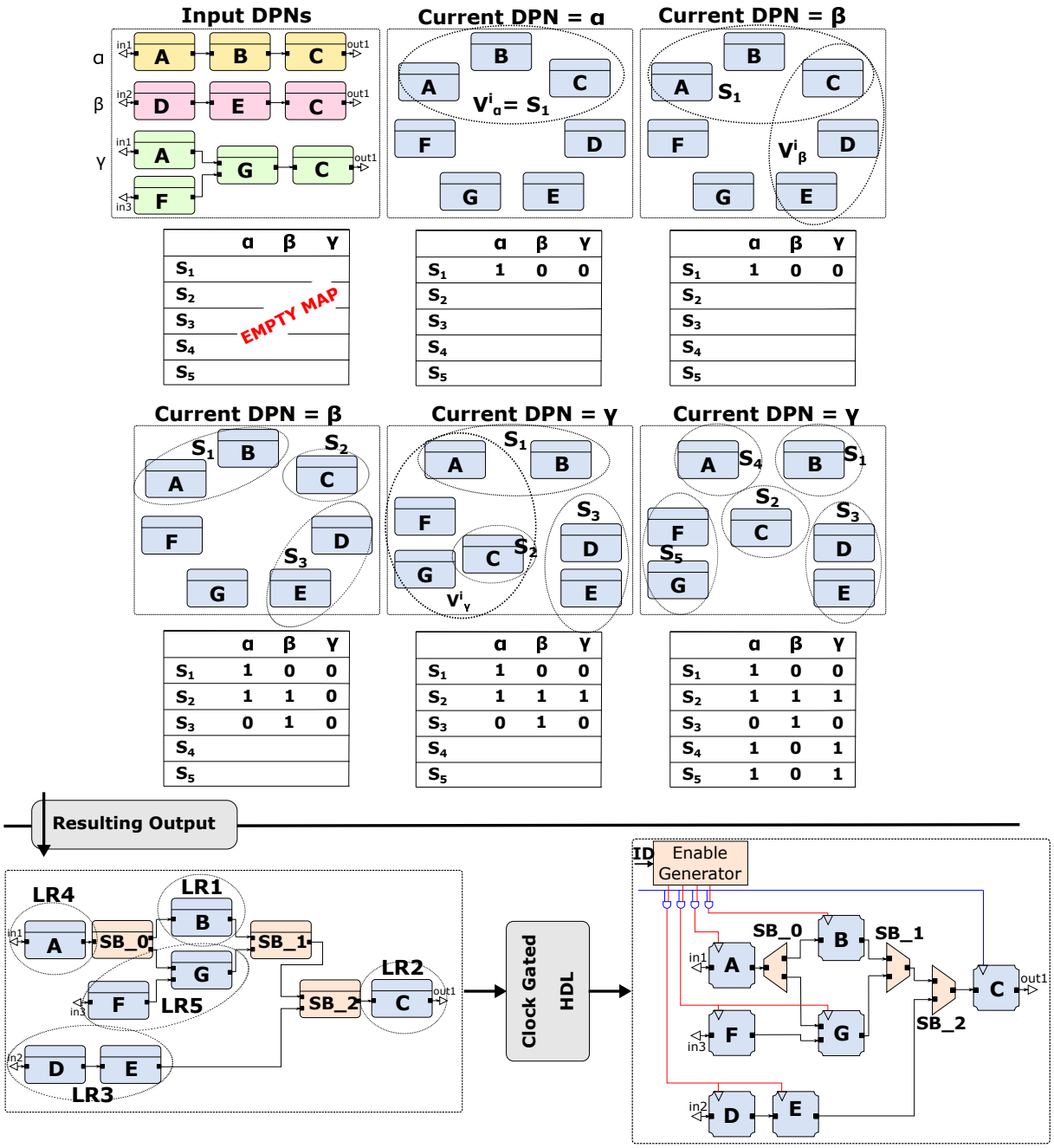


Figure 3.6: Logic Set Definer: a step-by-step example of Logic Regions identification and clock gating physical implementation .

Figure 3.6 shows also the resulting clock gated output of the proposed process. Five different *LRs* are identified from the three input dataflow specifications. Please note that the

SBoxes do not belong to any *LRs*. Indeed they are not synchronous elements and, in a clock gating perspective, they do not have any clock to switch off. The *Enable Generator* generates the enable signals for the clock gating cells on the basis of the requested functionality and it is automatically created and instantiated in the design.

3.4 Coprocessor Generator

MDC tool is able of automatically composing, synthesizing and deploying a runtime reconfigurable coprocessors compliant with Xilinx ISE Design Suite. Figure 3.7 provides an overview of the flow; MDC generates the CGR core as described in Section 3.1. Then, starting from the composed multi-dataflow (*Multi-flow IR*), MDC takes care of the IP generation, and its instantiation into a processor-coprocessor Xilinx architecture.

MDC uses the information from the high-level specification of the coarse-grain reconfigurable computing core to properly configure the coprocessing layer. Indeed, the CGR computing core is treated as a black box with a well-defined I/O interface characterized by: (1) the number of I/O ports, (2) the depth of the data channel of each I/O port and (3) the token pattern of each I/O port. In addition, to properly characterize the coprocessor also some configuration information are retrieved, as: (1) the ID whereby each input kernel is encoded and (2) which are the I/O ports required by the different input kernels.

Once collected the features of the generated multi-dataflow specification, MDC embeds the CGR computing core into a configurable template wrapper hereafter called Template Interface Layer (TIL). The TIL integrates a bank of configuration registers, to store the desired configuration, one (or more) front-end(s), to load data into the reconfigurable computing core, and one (or more) back-end(s), to read the computed data from the reconfigurable computing core. To easy deploy and use the the coprocessor, MDC provides also the Xilinx Vivado scripts to embed the logic into a processor-coprocessor architecture and the software drivers to ease its use.

3.4.1 Template Interface Layer

Generally speaking, coprocessing units can have different degrees of coupling with the host processor. A loosely coupled coprocessor is far from the processor, it is typically accessible through the system bus and it is affected by medium/high communication latencies for both control and data transfers. A tightly coupled coprocessor is closed to the processor, it has a dedicated full-duplex link and it often shares with the processor high-level memories. A loosely coupled coprocessor can be easily adopted in different contexts, since it is connected to a generic system bus. On the contrary, it is hard to extend the adoption of a tightly coupled coprocessor to different systems, since it has dedicated links and memory accesses.

MDC supports two different levels of applicability and coupling. User can choose between:

- a memory-mapped loosely coupled one, accessible through the system bus as a memory-mapped IP;
- a stream-based tightly coupled one, accessible through different full duplex links, one for each I/O port.

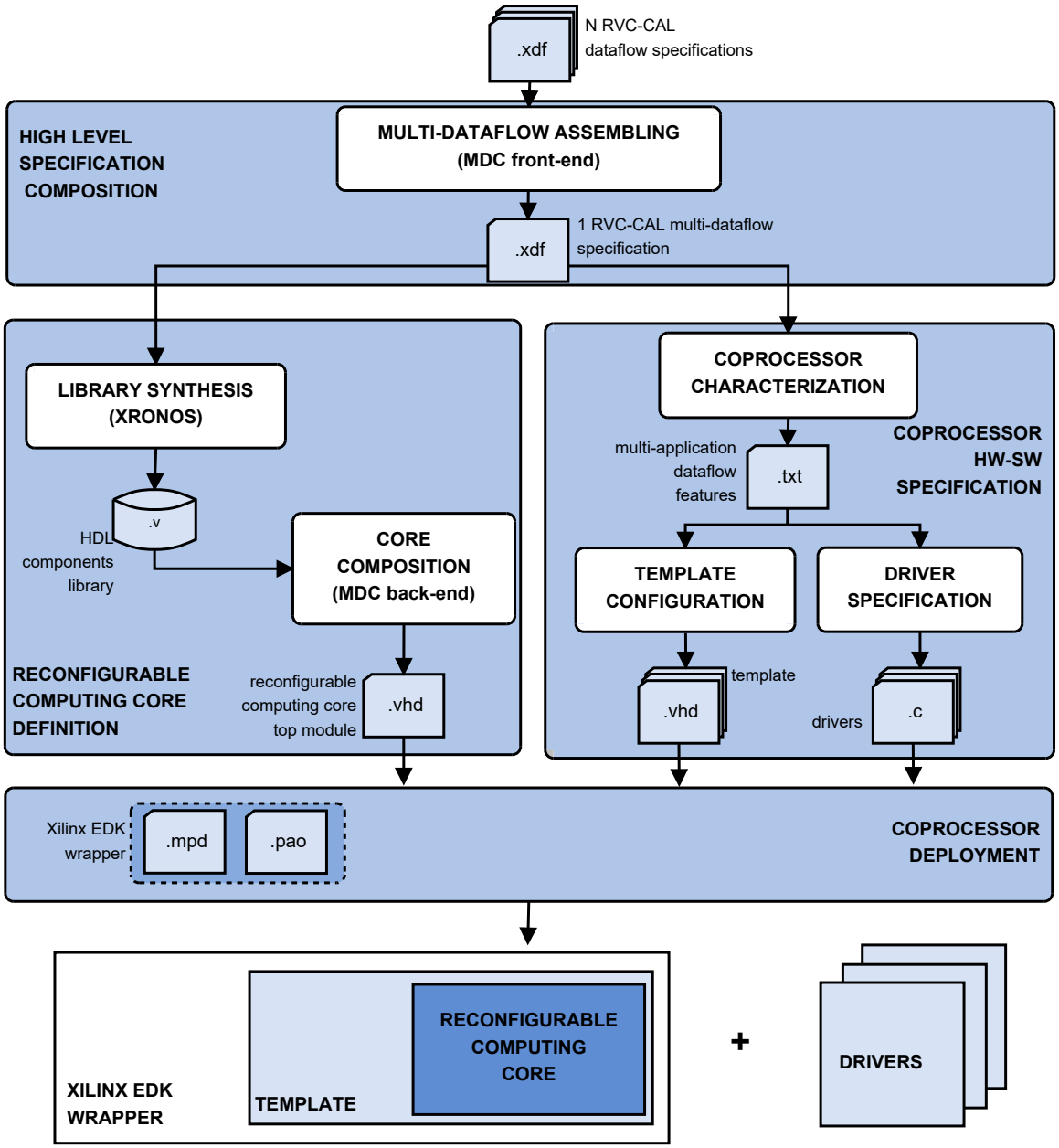


Figure 3.7: Coprocessor generator design flow overview.

The memory-mapped TIL (mm-TIL) is the easiest adaptable version of the automatically generated coprocessing layer. Figure 3.8 shows the architecture of the mm-TIL, main blocks are: the configuration registers bank and one local memory, one front-end and one back-end for each I/O port.

The local memory contains all the data to be processed by the coprocessor and the computed results. It has to be fully written by the processor before the coprocessor execution phase and it has to be fully read once the coprocessor has completed the task. A dedicated address range of the processor is reserved to the local memory. The configuration registers bank is the entity in charge of storing the configuration of the coprocessor. The configuration includes, a part the ID of the kernel to be executed, the base address, the data number for each I/O port and, only for the inputs, the burst size. The base address is the first address

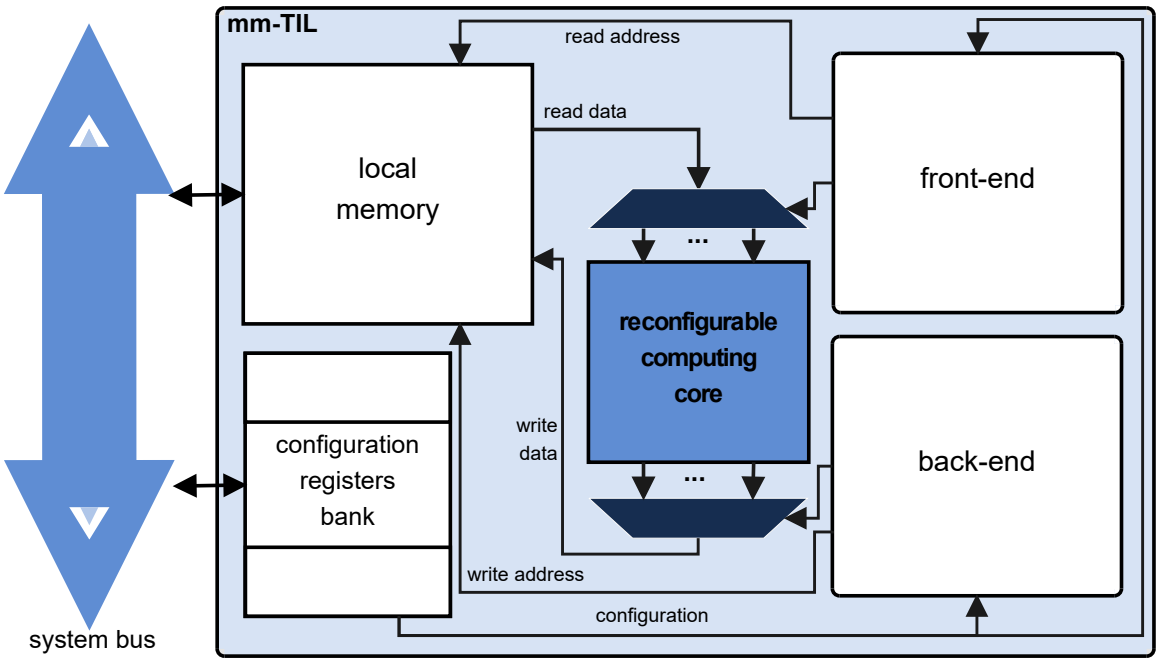


Figure 3.8: Architecture of the memory-mapped Template Interface Layer (mm-TIL).

of the local memory where the data have to be read/written. The data number is the amount of data to read/write from/to the local memory. The burst size is the number of data that have to be sent together to the reconfigurable computing core. The configuration registers bank holds an address range separated from the local memory one.

The front-end is responsible for the data transfer from the local memory to the reconfigurable computing core. Its execution flow can be divided in three different phases. At the beginning, an input port to load the data is chosen by means of a round robin priority policy. Then, the address of the local memory, where the related data is stored, is generated. Finally, a burst of data from the memory to the reconfigurable computing core is transferred. The front-end iterates on the described cycle until all the data have been transferred to the reconfigurable computing core.

The transfer of the processed data from the reconfigurable computing core to the local memory is performed by the back-end. Basically, the back-end architecture and execution flow are the same of the front-end ones, managing the output ports and dealing with memory writes instead of reads.

The stream-based TIL (s-TIL) is adopted to realize tightly coupled co-processing units. It is based on a Xilinx proprietary point-to-point connection protocol called Fast Simplex Link (FSL). The FSL is a very fast communication channel provided with FIFO memories, typically used within the Xilinx environment to connect host processors with hardware coprocessors. The s-TIL requires a different FSL channel for each I/O port. With respect to the mm-TIL the s-TIL can boost the coprocessor performance, not only leveraging on a faster communication channel, but also accessing in parallel different ports of the reconfigurable computing core. This parallelism can be fully exploited only if the host processor is able to read and write different FSL channels at the same time or if the co-processing unit is connected to different host processors.

Figure 3.9 depicts the s-TIL architecture. The main blocks are: the configuration registers

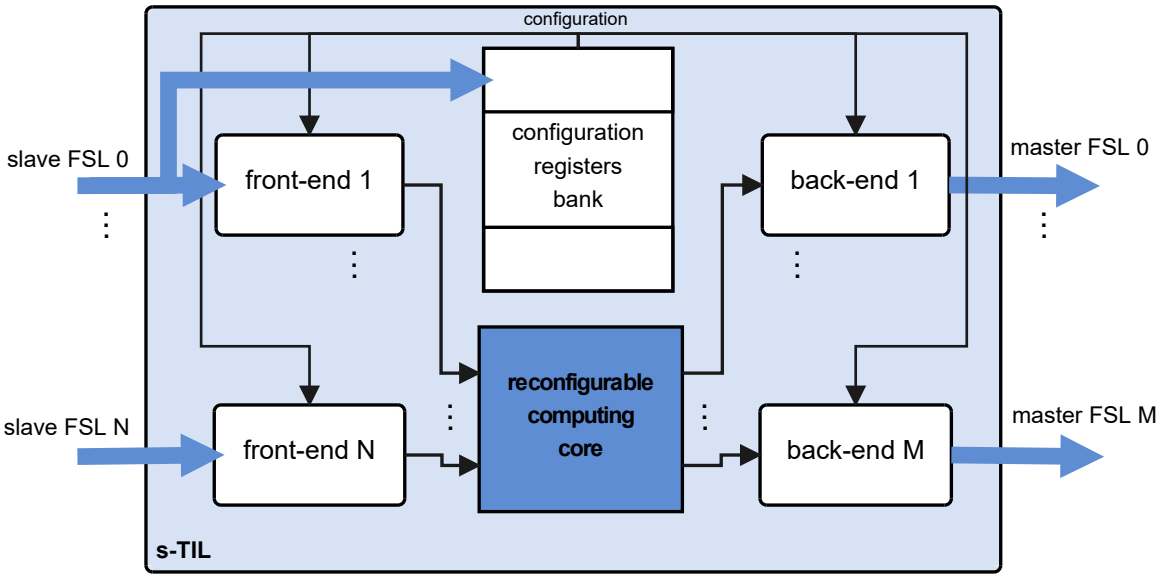


Figure 3.9: Architecture of the stream-based Template Interface Layer (s-TIL).

bank, one front-end per input port and one back-end per output port. The configuration registers bank, as in the mm-TIL, saves the coprocessor configuration. It is not mapped in a specific address range of the host processor, but it is supplied by one of the FSL links. On this latter, the control data transfers are differentiated from the processing ones by means of the control bit of the FSL protocol. When the control bit is high the data on the FSL are stored into the registers bank; whereas, when it is low the data are forwarded to the reconfigurable computing core. The base address configuration is no longer necessary: data come from (and have to be written) directly into a FSL channel instead of a local memory.

In the s-TIL the front-end and back-end aims are the same as in the mm-TIL: the front-end transfers data from an input FSL bus to the reconfigurable computing core and the back-end transfers data from this latter to an output FSL bus. In the s-TIL each I/O port is served by a different front-end or back-end, thus managing a point-to-point transfer of data from/to one of the FSL buses to/from one of the reconfigurable computing core ports (the port selection phase is no longer needed). No local memory is available; therefore, there is no address generation phase and the loading/storing phase of the different I/O ports is parallel.

3.4.2 Driver Specification

MDC, beyond providing the HDL gives also the software drivers to ease its use. In particular the features related to the configuration are used to specify the drivers, offering an interface that masks the system configuration complexity. MDC provides a C function for each configuration of the CGR coprocessor. For each functionality the user is required to only call the related function, and specify for each port the number of data to be read/written and their values.

The driver suite is split in two different levels:

- low level drivers (LLDs): manage the processor/coprocessor communication;
- high level drivers (HLDs): deal with the application issues, masking the system configuration complexity.

Listing 3.1: Low level coprocessor driver example.

```
...
// memory-mapped config writing
# define mmCOPR_write_config_<port_name>(int value) \
Xil_Out32(COPR_REGS_BASE_ADDR
+<port_name>_OFFSET, value);

// stream config writing
# define sCOPR_write_config_<port_name>(int value) \
cputfsl(value, FSL_ID_<port_name>);
...
```

LLDs encapsulate the system macros for writing/reading memory locations and for putting/getting data to/from the FSL buses into generic I/O functions (see Listing 3.1). The HLDs, starting from these generic I/O functions, manage the coprocessor configuration and data transfer. For each I/O port of the reconfigurable computing core, a configuration word is written into the proper coprocessor register. Then, for each input port involved in the current computation, a specific HLD primitive is used to send the data to be computed from the host processor to the coprocessor. At last, as the processor receives an interrupt from the co-processing unit, a specific HLD primitive is adopted to read back the results into the processor from the output ports (see Listing 3.2).

Listing 3.2: High level memory-mapped coprocessor driver example

```
...
// configuration
int config_<port_name> =
(size_burst_<port_name><<(SIZE_ADDR+SIZE_CNT))
| (size_<port_name><<SIZE_ADDR)
| base_addr_<port_name>;
mmCOPR_write_config_<port_name>(config_<port_name>);

// data sending
for(int i=0;i<size_<port_name>;i++)
mmCOPR_write_mem(base_addr_<port_name>
+ i*4, data_<port_name>[i]);

// data receiving
for(int i=0;i<size_<port_name>;i++)
data_<port_name>[i]=
mmCOPR_read_mem(base_addr_<port_name>
+ i*4);
...
```

No changes would be required if a different actors communication scheme is adopted. Indeed, it does not affect at all the processor/coprocessor communication scheme. Therefore, no changes at the drivers generation level are necessary to target a different scenario.

3.4.3 coprocessor Deployment

In the final step the peripheral is integrated and deployed as a standard Xilinx IP. The inputs are the HDL description of reconfigurable computing core, its front-end and its back-end; whereas, the output is the resulting Xilinx IP comprehensive of software drivers. The coprocessor is encapsulated through a Xilinx EDK wrapper to be included within the Xilinx IP

catalog. The wrapper is mainly composed of two files: the Microprocessor Peripheral Definition (MPD), used to define the peripheral interface within the system, and the Peripheral Analyze Order (PAO), needed to associate and properly order all the peripheral source files.

As well as in the driver case, no changes would be required if a different actors communication scheme is adopted. The final system integration phase is not affected by the internal actors communication scheme within the reconfigurable computing core, since the TIL template masks it.

Chapter 4

Coarse-Grain Reconfiguration on ASIC - Automating Power Gating from a Dataflow Representation

Modern embedded systems are required to accommodate different functionalities over the same substrate and provide flexibility at the hardware level. Coarse-grain reconfiguration (CGR) is a suitable solution to this need, being able of offering a certain degree of flexibility minimizing resource redundancy. However, CGR systems can still be power hungry and dedicated design frameworks could help in the efficient implementation of runtime low-power reconfigurable platforms. The main contribution of this chapter is the automatic implementation of a power gating strategy to CGR systems. Exploiting the MDC capability of identifying homogeneous logic regions within the system (i.e. set of resources that can be turned on and off together), this work extends the approach presented in Section 3.3 to achieve dynamic power management by means of coarse-grain power gating techniques. The work presented in this Chapter has been conducted in a collaboration between the Microelectronics and Bioengineering Lab (EOLAB) (University of Cagliari) and the Intelligent system DEsign and Applications (IDEA) Lab (University of Sassari), into the context of the "RPCT - Reconfigurable Platform Composer Tool" Project, funded by the Sardinian Regional Government under agreement *L.R. 7/2007, CRP-18324*.

4.1 State of the Art: Power Management in ASIC systems

CGR platforms offer high performance and flexibility, allowing the execution of a large set of applications over the same substrate [42] (see Section 2.1). In a CGR system all the logic necessary to implement different functionalities are present on the substrate, and different configurations are enabled by multiplexing the resources in time. This means that in every moment there are resources, not involved in the active configuration, that consume power. Several techniques (clock gating, multi-frequency, operand isolation, multi-threshold, multi-supply libraries, power gating, etc.) can be applied to reduce power consumption and, in

some cases, they are automatically implemented by commercial synthesis/place-and-route tools.

Clock gating is a really popular technique, able to reduce the dynamic power consumption due to the clock tree and to sequential logic up to the 40% [132]. It consists on shutting off the clock of the unused synchronous logic, and it can be applied at different granularities: fine-grain approaches act on single registers, whereas coarse-grain ones are referred to a set of resources. Clock gating has been deeply employed in application-specific integrated circuit (ASIC) designs for more than 20 years [84, 120]. Commercial synthesizer such as Cadence RTL Compiler (or the more recent Genus) [17, 18], or Synopsys Design Compiler [114] are able of gating groups of flip-flops that are enabled by the same control signal. Some works focussed on the application of clock gating at a higher level. In [78] authors described a systematic approach for computing the clock gating logic of synchronous sub-circuits described in HDL. In the dataflow field Bezati et al. [9] presented an extension of an High-Level Synthesis tool, Xronos, to selectively switch off clock signal for parts of the circuit that are idle due to stalls in the pipeline, to reduce power consumption. However, they target only FPGA. As described in Chapter 3, the MDC tool has the capability of identifying the minimum set of independent circuitry regions, applying to all of them clock gating for both FPGA and ASIC targets [82].

More complex power saving strategies such as, voltage/frequency scaling [44, 37] and power shut-off schemes [7, 53] can be extremely beneficial. In communication networks design, SONICS exploited many of traditional techniques for power management (from clock gating to voltage scaling etc.) to deploy an efficient interconnect [119]. The *ARM big.LITTLE* processor features two sets of processors optimized for different purposes that may be alternatively shut-off [53] when not used. And the PULPv2 Cluster [94] is split into two separate voltage and frequency domains. In the CGR architectures field Jafri et al. [52] adopted a dynamically reconfigurable resource array (DRRA), to assemble a CGR energy-aware architecture. They integrated a power management infrastructure, power management intelligence and architectural support for dynamic parallelism with the DRRA, to implement energy aware task parallelism. However, the integration of the above described power management strategies required manual intervention of the designer, and they could result in a complex, error prone and time consuming process.

Nowadays, some of the EDA companies offer the possibility of automatically integrating low power techniques such as clock gating, dynamic voltage/frequency scaling or power gating. A power format file allows designers to specify the power intent early in the design and without any direct modification of the RTL code. The two most commonly used low power flows are the Unified Power Format (UPF) [49] from Synopsis and the Common Power Format (CPF) [105], whose definition is driven mainly by developers using Cadence. The implementation, simulation and verification of low power designs adopting power format files have been widely studied in the last years. Kulkarni et al. [56] adopted a UPF-based flow to explore a power aware verification flow. In [72] authors studied the advantage of multi-VDD power reduction technique on the ISCAS89 S38417 benchmark circuit, adopting a CPF based ASIC design flow. Lopes et al. [66] presented the design and implementation of a CGR array for low-power biological signal processing. Authors adopted the UPF to integrate power gating in the presented CGR architecture. However, in these works the power format is manually defined. Manual definition of a power format file can be error prone and time consuming, and also not easily applicable to automatically generated CGR systems, such as the ones considered in this thesis.

Recently some works focused on the application of power saving methodologies, automatically generating a power format file. In [39] authors present a SCPower extension that allows to inject power specification into synthesizable hardware designs in SystemC language, providing the automatic generation of the UPF file. However, it is focused more on enabling power-aware verification of SystemC designs. Qamar et al. [91] present a methodology that consider the application of clock and power gating techniques to the register transfer level (RTL) generated automatically by high-level synthesis (HLS), using SystemC code. The designer can define the power intent directly at system level and add power management control logic to implement the low power methodology. At high level of abstraction they specify the power intent, to generate the CPF to implement the power gating. However, this work still requires hand-work. Indeed it mainly move the definition of the power intent from RTL level to higher level, specifying it through the insertion of `#pragmas` into the SystemC code. Furthermore the logic to be switched off through power saving techniques is not automatically identified. Macko [67] proposed a method for automation of power-management specification. The input of the method is a system functional model in SystemC and ESL simulation results in VCD. The output is an enriched system model, which includes the power-management specification using SystemC/PMS. However, this method is limited to SystemC high level description, and is not applicable to CGR systems.

Most of the above described strategies still require high hand-work, requiring the designer to identify the logic to be switched-off and in some cases also to specify the power intent. The work presented in this Chapter describe a methodology for the automatic application of the power gating technique to CGR systems, and its application into the Multi-Dataflow Composer (MDC) tool. Working at dataflow-level, the methodology identify the logic to switch-off, and generates accordingly a CPF file with the specification of the required power intent and the HDL files, containing the controller to properly enable/disable the power gating logic.

4.2 Methodology

The goal of power gating is to minimize the consumption of static power. The main idea behind it is: if a specific portion of the design is not used in a given computation mode, then it can be completely powered-down by means of a sleep transistor. This technique, as the clock gating one, is applicable at different granularities: fine-grain approaches require to drive a different sleep transistor for every cell in the system, while coarse-grain ones, again, operate on a set of resources instantiating one sleep transistor to drive different cells connected to a shared power network. However, clock gating can be handled almost easily during the design and implementation process; power gating is more invasive technique, since it requires the insertion of several extra logic to handle the inter-block communication and the powering down/up transitions.

Firstly, it is required the insertion of the *sleep transistors* (or *power switches*) between the gated region (or *power domain*) and the main power supply to switch on/off the derived power supply. However, this is not enough to handle the correct power-down/up sequence, which includes also the isolation on signals from the shut-down domain. The power domains to be powered-down have to be isolated before power is switched off, and have to remain isolated until the power is again totally on. The *isolation logic* is typically used between the powered-down region and the powered on ones, to avoid the transmission of spurious

signals in input to ON-cells. In certain cases, the state of control flops need to be maintained to guarantee the proper operation of the system, when the regions are powered-on. For this purpose, the *state retention logic* is adopted. Retention cells typically have a low power consumption shadow register, connected to the main power supply, where to save the state of the main register when the domain is powered-down.

This additional logic can be manually inserted by the designers in the RTL architecture or through a power format file. Manual definition is highly error prone: it requires modelling the impact of low-power during simulation and providing multiple definitions for synthesis, placement, verification and equivalence checking [90]. In this work, the CPF is adopted. The CPF can be divided into two main parts: technology and power intent.

- The technology part sets the timing libraries and specifies the low-power cells to be used within the technology specific physical libraries.
- The power intent part depends on the design and manages the power domains, the power modes and the respective transitions. Here the power cells, specified on the technology part, are instantiated and associated to the logic in the design, accordingly with the power domains to which they belong.

4.2.1 Automatic Power Gating Implementation

The first step for the automatic implementation of a power gating strategy requires the identification of the logic blocks to be switched off (power domains - PDs). The assumption is that, once identified the minimum set of Logic Regions (*LRs*) composed of actors always active/inactive together, these *LRs* can be powered-down. Thus the Algorithm 1 (see Appendix A) used by MDC to identify the *LRs* can be exploited also to identify the switchable PDs.

However, MDC'*LRs* for clock gating applications do not include all of the logic that can be switched off through power gating. Since power gating technique acts on both the static and dynamic power consumption, all of the FUs, included the combinatorial ones, have to be considered. Thus, in the perspective of applying power gating to the to CGR systems generated by MDC, also the *SBoxes* have to be included within the *LRs*. [1] Thus, the identification process described in Section 3.3 is here extended to include also the *SBoxes*.

Considering $DFG\langle V, E\rangle$ as the directed graph of the multi-functional specification, when power gating strategies are implemented, the V set includes both the set of computational actors, V', and the set of *SBoxes*, V'', as $V = V' \cup V''$. The first step for the identification of the power domains applies the Algorithm 1 presented in Section 3.3 to identify the basic set of *LRs*. Then, the *SBoxes* identification Algorithm 3 (see Appendix B) is applied. This Algorithm exploits the information within the configuration table (C_TAB) to add the *SBoxes* V'' to the partition S. The algorithm consists of two main steps:

- for each *SBox*, the algorithm creates a set DPN_{SBi} analysing all the input DPNs. If a DPN activates the considered *SBox* (the corresponding value within the C_TAB is not a X) then it has to be active when that functionality is requested. Therefore, the input DPN is added to DPN_{SBi}. When all input DPNs have been cross-checked, the DPN_{SBi} set is added to the SB_MAP. This latter is a map used to keep trace for each *SBox*, SB_i, of the set of DPNs that need its activation during the execution.

[1] In a purely clock gating based methodology, this was not necessary since they do not have (in the current MDC implementation) any clock to be switched-off.

- For each SB_i in the SB_MAP, the algorithm compares the value DPN_{SBi} with all the N *LRs* in the LR_MAP. The following situations may occur:
  - the considered *SBox* set is activated by the same DPNs triggering the considered *LR* (DPN_{SBi}=$DPNs_j$): SB_i does not represent a new region and it is simply added to S_j;
  - the considered *SBox* set is activated by different DPN(s) with respect to all the considered *LRs* (SB_i is still unassigned, $!assignedSB = 1$): SB_i represents a new region; therefore, a new set ($S_N = SB_i$), is added to LR_MAP.

At the end of the process, each *LR* can be mapped into a different PD. This implies creating, for each *LR*, a power domain into the CPF file, defining for each switchable PD also the shut-off condition. Instances belonging to each PD are related to the actors that belong to the corresponding *LR*.

Considering an example with two input DPNs (*DPN1* and *DPN2*), were three *LRs* are identified (*LR1* shared by the two DPNs, *LR2* used only by *DPN1* and *LR3* used only by *DPN2*), three PDs can be created in the CPF. The following CPF line depict the creation of the three PDs. PD_AO is the default always on domain, corresponding to *LR1*. While PD1 and PD2 are the switchable PDs corresponding to *LR2* and *LR3*.

```
# create always on power domain corresponding to LR1
create_power_domain -name PD_AO -default {...}

# create switchable power domain corresponding to LR2
create_power_domain -name PD1 -instances {...}\
-shutoff_condition {switch_en1} -base_domains {PD_AO}

# create switchable power domain corresponding to LR3
create_power_domain -name PD2 -instances {...}\
-shutoff_condition {switch_en2} -base_domains {PD_AO}
```

Then, for each considered DPN a different Power Mode (PM) can be created. Each PM represents a steady state of the design in which some PDs are active, while others are disabled, according to the *LRs* to be activated by the requested functionality. Following the example with two input DPNs and three *LRs*, three PMs can be defined in the CPF: one default PM, where all of the PDs are powered-on (PMdef), one corresponding to DPN1 (PM1), and one corresponding to DPN2 (PM2).

```
# create default power mode with all PDs active
create_power_mode -name PMdef\
-domain_conditions {PD_AO@on PD1@on PD2@on} -default

# create default power mode corresponding to DPN1
create_power_mode -name PM1\
-domain_conditions {PD_AO@on PD1@on PD2@off}

# create default power mode corresponding to DPN12
create_power_mode -name PM2\
-domain_conditions {PD_AO@on PD1@off PD2@on}
```

Finally, for each switchable PD isolation and state retention logic can be defined.

```
# create rules for isolation logic insertion
create_isolation_rule -name iso1 -from PD1\
```

```
-isolation_condition {iso_en1} -isolation_output low
create_isolation_rule -name iso2 -from PD2\
-isolation_condition {iso_en2} -isolation_output low

# create rules for state retention insertion
create_state_retention_rule -name st1 -domain PD1\
-restore_edge {rstr_en1} -save_edge {save_en1}
create_state_retention_rule -name st2 -domain PD2\
-restore_edge {rstr_en2} -save_edge {save_en2}
```

4.2.2 Step-by-step example

Starting from the output of the step-by-step example in Figure 3.6, Figure 4.1 illustrates an example where Algorithm 3 is applied. The SBoxes involved are: SB_0, SB_1 and SB_2. At first the algorithm processes C_TAB to create SB_MAP:

1. SB_0 is considered.
   - Create a new empty set DPN_{SB0}
   - Analyse all $DPNs$ of SB_0 in C_TAB
     - $SB_0(DPN_\alpha) = 0$: add α to DPN_{SB0}.
     - $SB_0(DPN_\beta) = X$: do not add β to DPN_{SB0}.
     - $SB_0(DPN_\gamma) = 1$: add γ to DPN_{SB0}.
   - DPN_{SB0} is added as value to key SB_0 in SB_MAP.
2. SB_1 is considered.
   - Create a new empty set DPN_{SB1}
   - Analyse all $DPNs$ of SB_1 in C_TAB
     - $SB_1(DPN_\alpha) = 0$: add α to DPN_{SB1}.
     - $SB_1(DPN_\beta) = X$: do not add β to DPN_{SB1}.
     - $SB_1(DPN_\gamma) = 1$: add γ to DPN_{SB1}.
   - DPN_{SB1} is added as value to key SB_1 in SB_MAP.
3. Finally SB_2 is considered.
   - Create a new empty set DPN_{SB2}
   - Analyse all $DPNs$ of SB_2 in C_TAB
     - $SB_2(DPN_\alpha) = 0$: add α to DPN_{SB2}.
     - $SB_2(DPN_\beta) = 1$: add β to DPN_{SB2}.
     - $SB_2(DPN_\gamma) = 0$: add γ to DPN_{SB2}.
   - DPN_{SB2} is added as value to key SB_2 in SB_MAP.

Once the SB_MAP is determined, Algorithm 3 compares each DPN_{SBi} in SB_MAP with all the S_j in LR_MAP.

1. SB_0 is considered.

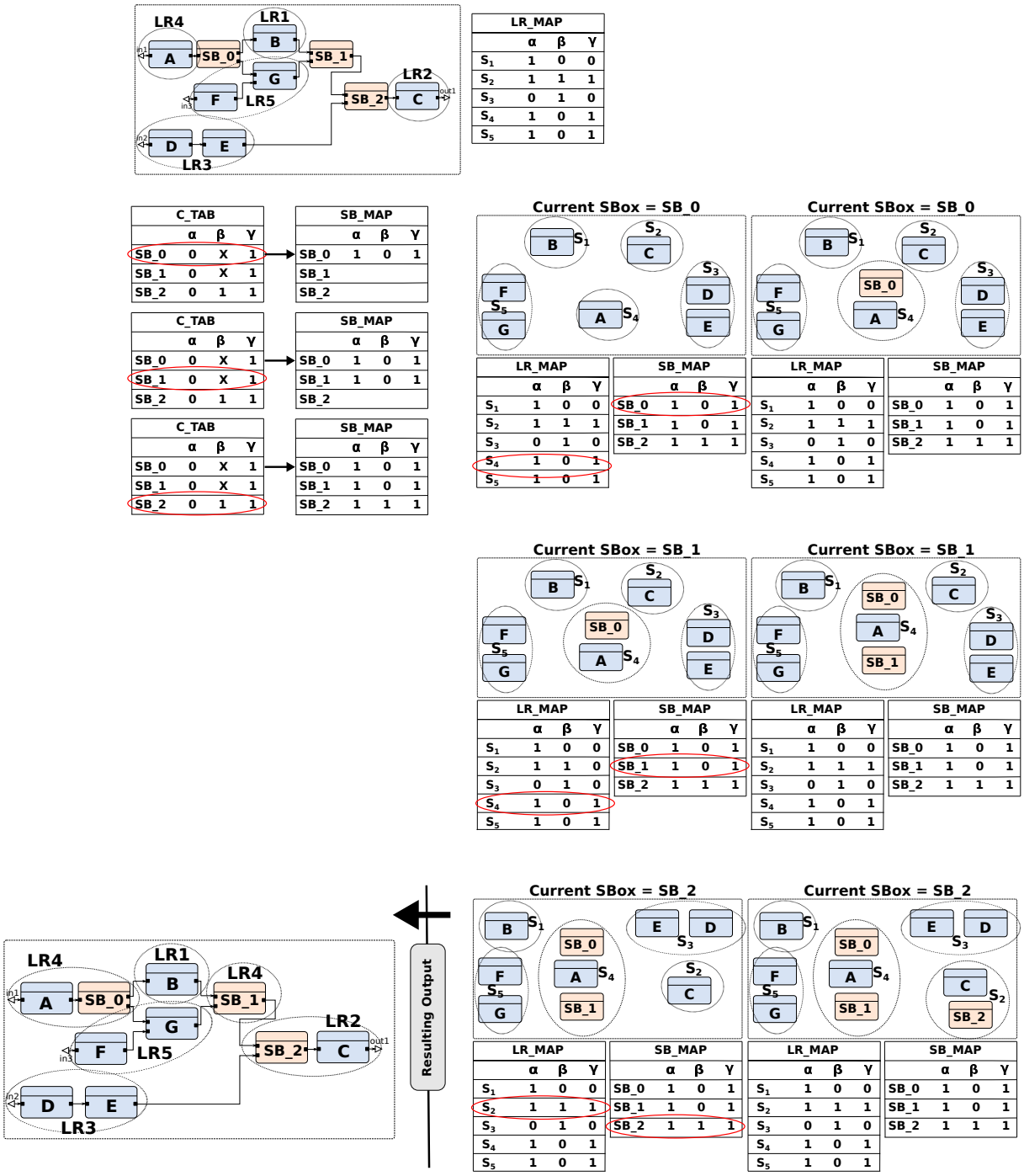


Figure 4.1: Logic Set Definer: a step-by-step example of the Logic Regions set extension with the SBoxes.

- Compare DPN_{SB0} with all sets S_j in LR_MAP
  - $DPN_{SB0} = DPN_{S4}$
  - SB_0 is added to $LR\,S_4$ in LR_MAP

2. Then SB_1 is considered.

- Compare DPN_{SB1} with all sets S_j in LR_MAP

- $DPN_{SB1} = DPN_{S4}$
- SB_1 is added to $LR\,S_4$ in LR_MAP

3. Finally SB_2 is considered.

   - Compare DPN_{SB2} with all sets S_j in LR_MAP
     - $DPN_{SB2} = DPN_{S2}$
     - SB_2 is added to $LR\,S_2$ in LR_MAP

4.3 Integration in MDC

The MDC *Power Manager* has been extended to provide, in addition to the clock gating technique, also the the power gating technique. The Algorithm 1 for the identification of the *LRs* has been extended with Algorithm 3, to identify the best set of *LRs* according to the saving technique selected but the user. Figure 4.2 shows the MDC design flow that includes the power management extension.

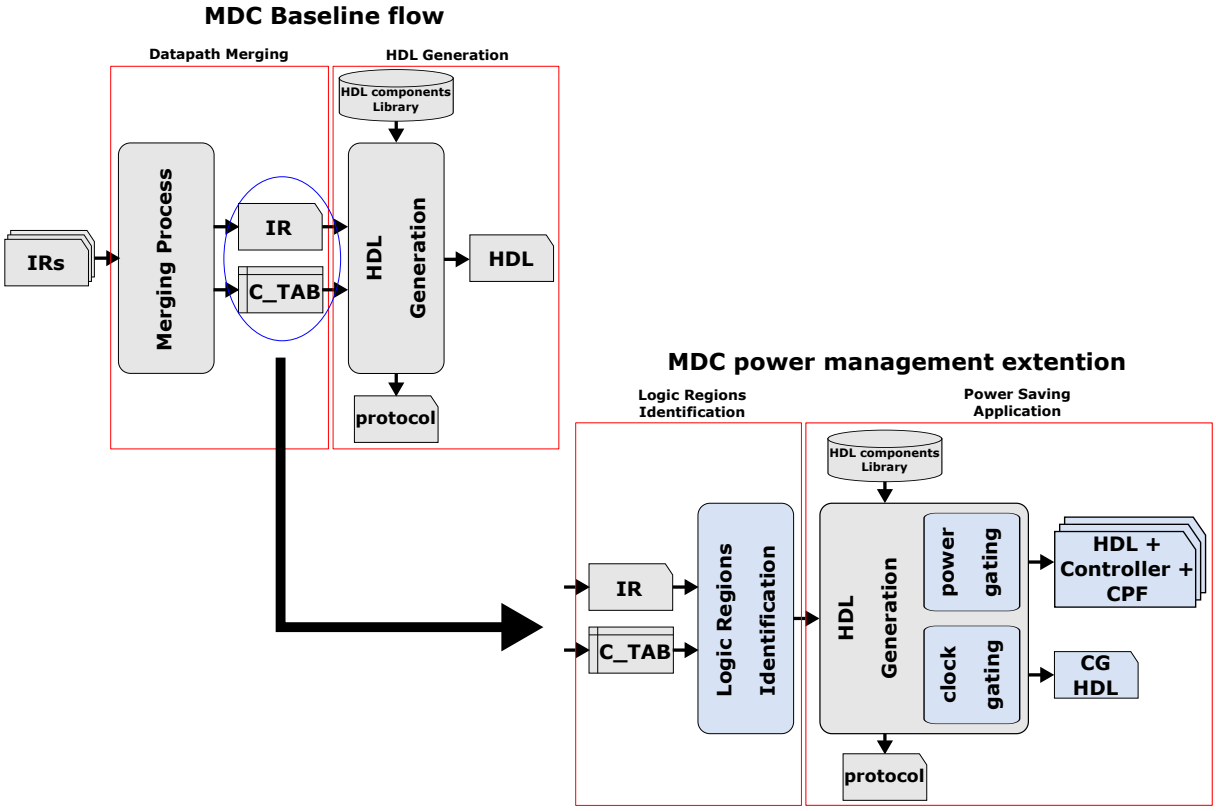


Figure 4.2: MDC design suite: baseline flow (on top) and corresponding power extension (on bottom).

In order to give to the synthesizer the information about the power intent, MDC provides also a CPF file. The CPF generated by MDC is divided in two parts, following division described in Section 4.2.

- technology part: is dependent on the adopted technology. Here the timing libraries (`define_library_set`) and low-power cells (`define_xxx cell` commands) are specified;
- power intent part: depends on the design and manages the PDs, PMs and low power rules. The PDs are created through the `create_power_domain` command, and for each switchable PD the associated FUs from the design are specified. All the logic belonging to the always-on domain is automatically associated to the default PD (PDdef), which has no associated switching-off logic. Then, for each switchable domain also the isolation (`create_isolation_rule`) and retention rules (`create_state_retention_rule`) are specified.

An excerpt of the power intent CPF file corresponding to the five *LRs* identified in Figure 4.1 is provided hereinafter.

```
#####################################################
# CPF file automatically generated by:
# Multi-Dataflow Composer tool
#####################################################
set_cpf_version 2.0
set_hierarchy_separator /

#####################################################
# Technology part of the CPF
#####################################################

# define the library sets
define_library_set -name set1_wc\
-libraries {lib1_wc lib2_wc}
define_library_set -name set1_bc\
-libraries {lib1_bc lib2_bc}

# define the isolation cells
define_isolation_cell -cells {ISOL*} -enable ISO\
-power VDD -ground VSS  -valid_location on

# define the always on cell
define_always_on_cell -cells "AO_BUF1 AO_BUF2 AO_INV1 AO_INV2 ..."\
-power_switchable VDD -power TVDD -ground VSS

# define the state retention cell
define_state_retention_cell -cells RTN_FF* \
-power_switchable VDD -power TVDD -ground VSS\
-save_function SAVE -restore_function !NRESTORE

# define the power switch cells
# define the power switch cells
define_power_switch_cell -cells "HD_SW1*" \
-stage_1_enable NSLEEPIN1 -stage_1_output NSLEEPOUT1\
-stage_2_enable NSLEEPIN2 -stage_2_output NSLEEPOUT2\
-type header -power_switchable VDD -power TVDD

#####################################################
# Design part of the CPF (Power Intent part)
#####################################################
```

```
#identify the design for which the CPF is created
set_design top_module

# create power domains
create_power_domain -name PDdef -default # (inst_C sbox_2)
create_power_domain -name PD1 -instances {inst_B}\
-shutoff_condition {!powerController_0/pw_switch_en1}\
-base_domains {PDdef}
#repeat for all switchable domains

# create nominal conditions
create_nominal_condition -name off -voltage 0
create_nominal_condition -name on -voltage 1.1

# create power modes
create_power_mode -name PMdef\
-domain_conditions {PDdef@on PD1@on PD3@on PD4@on PD5@on}\
-default
create_power_mode -name PM1\
-domain_conditions {PDdef@on PD1@on PD3@off PD4@on PD5@off}
#repeat for all power modes

# associate library sets with nominal conditions
update_nominal_condition -name on -library_set set1_wc

# create rules for isolation logic insertion
create_isolation_rule -name iso1 -from PD1\
-isolation_condition {powerController_0/iso_en1}\
-isolation_output low
#repeat for all necessary isolation rules

# create rules for state retention insertion
create_state_retention_rule -name st1 -domain PD1\
-restore_edge {!powerController_0/rstr_en1}\
-save_edge {powerController_0/save_en1}\
-target_type both -secondary_domain PDdef
#repeat for all necessary retention rules

...

# indicate when the power intent for the design ends
end_design
```

To properly drive the low power logic, MDC adds to the reconfigurable HDL an automatically generated *Power Controller*, which is composed of a different finite state machine for each PD. The *Power Controller* properly drives: *a)* the enable of the *Power Switches* and of the *Isolation* cells, *b)* the save command of the *State Retention* cells (to store registers values before their shutting off), and *c)* the restore command of the *State Retention* cells (to retrieve registers values once switched on).

Figure 4.3 depicts an example of the finite state machine (FSM) that controls the low power logic. At the beginning the PD is active, and the FSM stays in the *IDLE ON* (S_1) state until the PD is enables (*en*). When the PD is disables (*!en*) the FSM starts the proper power-down sequence by isolating the PD (S_2), switching off the clock to guarantee the proper saving of the retention registers (S_3), saving the content of the retention registers (S_4) and finally

switching off the power (S_5). Then the FSM goes to an *IDLE OFF* state (S_6) where it remains until the PD is disabled (*!en*). Once the PD is activated again, the FSM starts the power-up sequence switching on the power (S_7), restoring the state from the state retentions (S_8), enabling the clock (S_9) and removing the isolation (S_{10}), returning finally to the initial *IDLE ON* state (S_1).

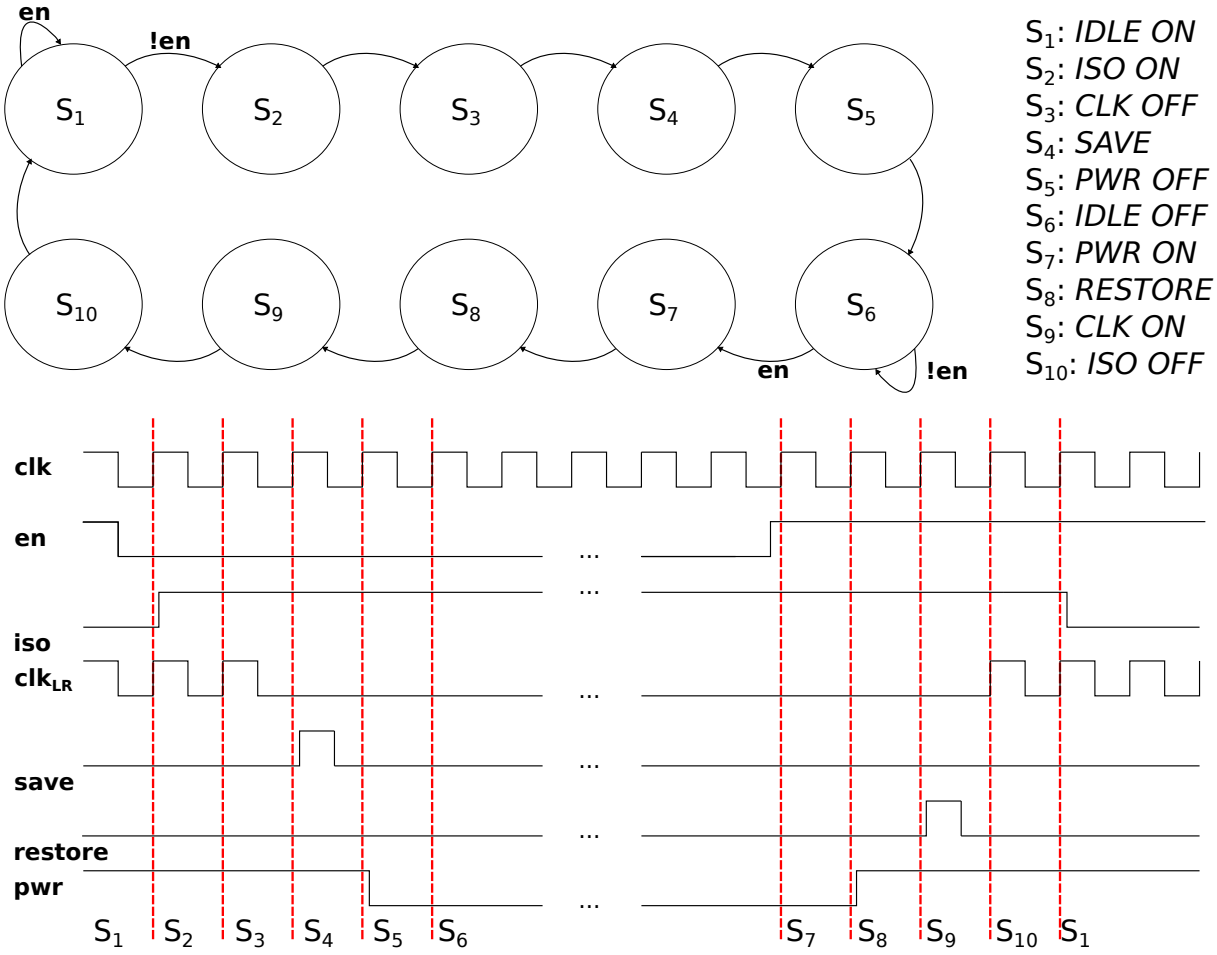


Figure 4.3: Finite state machine that controls the power logic belonging to a switchable PD.

Figure 4.4 shows the implementation of the power gating management in the resulting platform:

- *LR2* is shared by all of the DPNs and does not require any power management support. Thus, is is placed by default into the always-on domain PDdef (not indicated in figure);
- PD1, PD3, PD4 and PD5, corresponding to *LR1*, *LR3*, *LR4* and *LR5*, are power gated by means of three different types of cells, *Power Switch*, *Isolation* and *State Retention* cells (SBoxes do not need these latter since they are state-less);
- the *Power Controller* to properly generate the enable/restore signals for the power gating cells. For the proper operation of the state retention cells these PDs are also clock gated.

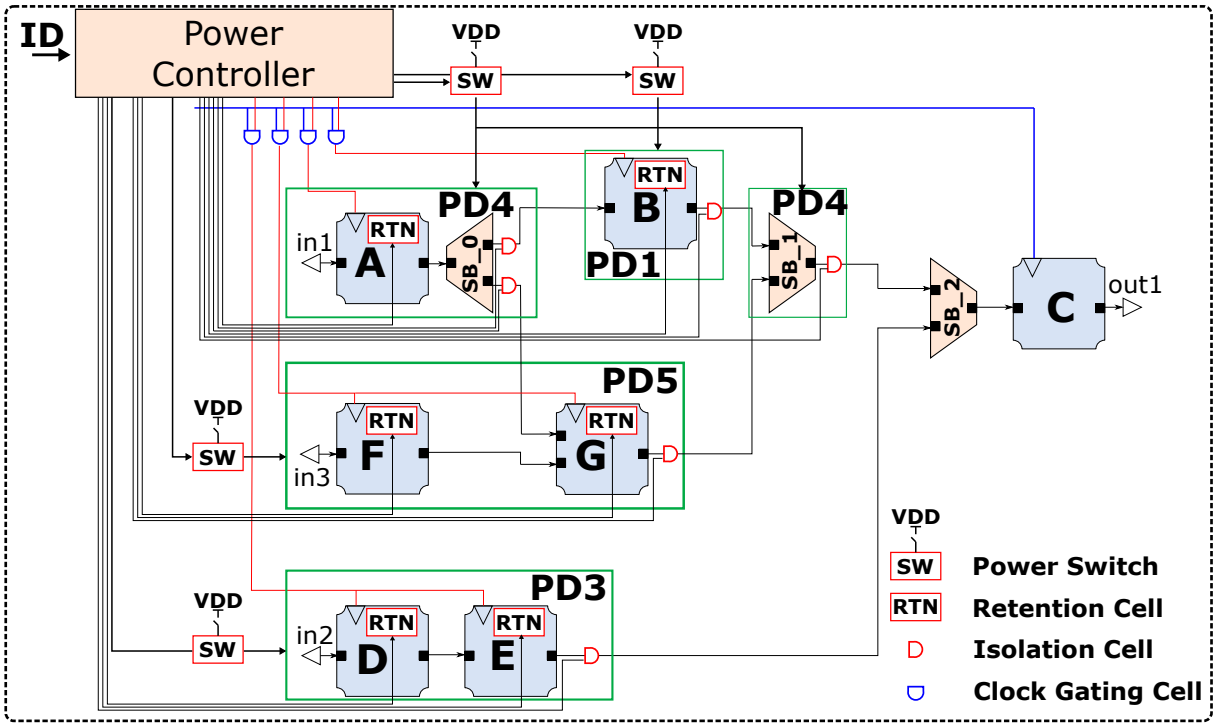


Figure 4.4: Logic Set Definer: power gating physical implementation.

4.4 Assessment

This section validates the proposed dataflow-based power management strategy in three different application scenarios for image processing:

- Spatial Anti-Aliasing (UC1), to correct the distortion effects of an image downsampled violating the Nyquist constraint;
- Zoom (UC2), to scale an image by a given zooming factor interlacing pixels of the original image with other pixels coming from adaptive interpolations of the neighbouring ones;
- Deblocking Filter (UC3), to remove image distortions, like blocking and ringing, coming from compression processes.

The applications have been profiled to identify the most computationally intensive code segments, called in the follow *computational kernels* or simply *kernels*, that require to be accelerated. For UC1, UC2 and UC3, six, seven and eight kernels respectively have been identified and modelled as DPNs according to the RVC-CAL dataflow model. The final intent is accelerating in hardware these applications by mapping their respective kernels over a different reconfigurable multi-functional architecture, assembled with MDC.

Table 4.1 depicts an overview of the kernels composition, functionalities and occurrences within the three applications. The number of actors is a raw index of the kernel size, which is important to understand the kernel power consumption results. A more precise insight of the size of each kernel is provided in Table 4.2 that reports both the overall kernel area - *column area* [μm^2] - (along with its percentage with respect to the corresponding use case)

Table 4.1: Computational kernels of the adopted use cases.

app	kernel	# actors	# occ	data size	functionality
UC1	*sorter*	2	18659	25	vector sorting
	min_max	1	14864	2	maximum/minimum finding
	rgb2ycc	19	256	3x4x4	RGB to YCrCb colour conversion
	ycc2rgb	18	256	3x4x4	YCrCb to RGB colour conversion
	abs	1	124366	1	absolute value calculation
	corr	10	2321	25	vector correlation
UC2	*min_max*	1	15142	2	maximum/minimum finding
	abs	1	70045	1	absolute value calculation
	sbwlabel	17	966	16	edge block checking
	chgb	7	3072	4	bilevel/grayscale block checking
	cubic_conv	6	341	16	cubic filter convolution
	median	9	1253	4	median calculation
	cubic	10	1496	4	linear combination calculation
UC3	*min_max*	1	32806	2	maximum/minimum finding
	filter	13	2235	10	vector filtering
	clip	2	346	2	vector comparison
	inner	9	1108	4	vector weighting and sum
	mdiv	7	346	4	vector biasing
	sign	1	346	1	sign calculation
	rgb2yuv	19	256	3x4x4	RGB to YUV colour conversion
	yuv2rgb	18	256	3x4x4	YUV to RGB colour conversion

and the area they share with other kernels - *column shared area* - (along with its percentage with respect to the overall kernel area). Last column reports, for each kernel, the percentage of its shared area in the considered use-case.For instance, in UC1, the 32.86% of *sorter* kernel is shared with other kernels belonging to UC1, while in the case of *min_max* the whole kernel (100% of the kernel) is shared with other kernels belonging to UC1.

Back to Table 4.1, the data size is referred to the number of tokens that the kernel is able to process for every execution. This metric gives an idea of the kernel complexity and of its execution latency. The occurrences of a given kernel (*#occ* in Table 4.1) are intended as the number of times the kernel is executed while running its corresponding application. If a kernel is executed several times, its activation frequency is high and, in turn, its switching activity largely impacts on the dynamic power consumption.

4.4.1 Assessment Setup

In terms of validation the proposed methodology is assessed over different coprocessor architectures, accelerating in hardware the applications presented in the previous section. Each coprocessor has been practically assembled with MDC, enabling/disabling the presented power-management extensions. For each of the presented UC, four different design are assembled:

- *UC* - assembled with the baseline version of MDC, without the adoption of any partic-

Table 4.2: Area occupancy of the kernels within the adopted use cases targeting a 90 nm ASIC technology. [* Percentages wrt to the UC total area (baseline row of Table 4.4); ** Percentages wrt the kernel total area.]

app	kernel	area [μm^2]	% *	shared area [μm^2]	% *	% **
UC1	*sorter*	12010	12.90	3946	4.24	32.86
	min_max	3946	4.24	3946	4.24	100.00
	rgb2ycc	37084	39.82	13955	14.98	37.63
	ycc2rgb	39862	42.80	13955	14.98	35.01
	abs	2089	2.24	2089	2.24	100.00
	corr	31956	34.31	10820	11.62	33.86
UC2	*min_max*	3919	3.33	3919	3.33	100.00
	abs	2089	1.78	2089	1.78	100.00
	sbwlabel	52943	45.01	10831	9.21	20.46
	chgb	19077	16.22	12579	10.69	65.94
	cubic_conv	20613	17.52	12444	10.58	60.37
	median	23782	20.22	16160	13.74	67.95
	cubic	21728	18.47	15478	13.16	71.24
UC3	*min_max*	3943	2.72	3943	2.72	100.00
	filter	55921	38.60	15083	10.41	26.97
	clip	6988	4.82	3943	2.72	56.43
	inner	23302	16.09	15911	10.98	68.28
	mdiv	19346	13.35	11279	7.79	58.30
	sign	842	0.58	0	0.00	0.00
	rgb2yuv	51195	35.34	20816	14.37	40.66
	yuv2rgb	51696	35.69	33815	23.34	65.41

ular power saving methodology.

- *UC_auto* - assembled with the baseline version of MDC and implementing fine-grain (register level) clock gating with the automatic support offered by the adopted commercial synthesizer.
- *UC_cg* - assembled with the clock gating extension of MDC.
- *UC_pg* - assembled with the power gating extension of MDC.

In all of the cases, the ASIC synthesis explorations have been performed using Cadence RTL Compiler. With respect to power consumption, all the power estimations have been extracted with RTL Compiler taking into account the real switching activity, collected during post-synthesis hardware simulations (performed with Cadence SimVision Debug) within VCD files. Therefore, dynamic power consumption numbers are accurate and representative of the on-off system conditions. The presented designs have been synthesized all at the same frequency, 200 MHz, targeting two different ASIC technology nodes: a 90 nm CMOS one (Section 4.4.2 and Section 4.4.3) and a 45 nm one (Section 4.4.4). The frequency choice has been determined by the kernels maximum achievable frequency (202.1 MHz for UC1 and UC3).

Table 4.3 shows the composition of the different use cases in terms of *LRs*. The *UC_cg* designs, in all the considered use cases, present a smaller number of *LRs* to be managed. When power gating is required, the *Logic Set Definer* adds also asynchronous sets of SBoxes, resulting then in a larger *LRs* number to be supported and managed.

Table 4.3: Use Case composition in terms of *LRs*

	LRs	
Use Case	UC_cg	UC_pg
UC1	9	14
UC2	13	19
UC3	15	22

4.4.2 90 nm CMOS Technology: complete power gating support

This section discusses the synthesis results achieved with a 90 nm CMOS technology. Table 4.4 summarizes, for each design, area occupancy and power consumption. The reported data are comprehensive of the overhead due to the specific power saving technique, such as the *AND* gates for the MDC-based clock gating implementations and the power controller, the isolation and the retention cells for the MDC-based power gating ones. All the power results are calculated as the sum of single kernels contribution multiplied by the number of their occurrences (reported in Table 4.1 for each application), divided by the sum of the occurrences of all the kernels[2]. This weighed average is representative of the designs power consumption during each use case typical execution.

As expected, both clock gating and power gating methodologies provide higher performance with respect to the baseline designs. For all the use cases the former are always above the 90% of saving, while the latter reach more than the 88%. MDC power saving methodologies, acting at the *LRs* level rather than at the register one, can achieve better results with respect to those achievable with fine-grain clock gating automatically applied by SoC Encounter. For all the use cases, with respect to *UC_auto*, the *UC_cg* designs are always above the 88% of saving, while *UC_pg* ones reach more than the 86%. In particular, for UC2 (Zoom) coarse-grain clock gating allows the saving of 91.27% of the total power consumption with respect to the automatic fine-grain clock gating design.

With respect to the area, the summary proposed in Table 4.4 confirms that power gating is an invasive technique, involving a severe area overhead. This latter is between 77.41% and 88.38% with respect to the baseline designs and more than the 94% with respect to the automatic fine-grain clock gating ones. This is due to the fact that, on the one hand, more *LRs* have to be handled with respect to a clock gating based methodology (as reported in Table 4.3) while, on the other, power gating implies adding and managing more components than simple clock gating strategies).

Table 4.5 better focuses on power results analysing separately, for all the considered use cases and designs, the static power consumption and the dynamic one. As expected, power gating demonstrates higher performance in terms of static power consumption. It allows

[2]Kernel contribution means the power consumption of the design when the considered kernel is enabled.

Table 4.4: 90 nm ASIC synthesis results. [* Percentages wrt to the baseline design without power-management; ** Percentages wrt to the baseline design implementing fine-grain clock gating with SoC Encounter]

Design	Area			Power		
	$[\mu m^2]$	%*	%**	$[\mu W]$	%*	%**
UC1	93136	—	—	5030.54	—	—
UC1_auto	84925	-8.82	—	3553.15	-29.37	—
UC1_cg	93435	+0.32	+8.82	376.08	-92.52	-89.42
UC1_pg	165229	+77.41	+94.56	490.57	-90.25	-86.19
UC2	117627	—	—	6181.85	—	—
UC2_auto	105972	-9.91	—	4995.84	-19.19	—
UC2_cg	118007	+0.32	+11.35	436.05	-92.95	-91.27
UC2_pg	221589	+88.38	+109.09	619.69	-89.98	-87.60
UC3	144863	—	—	7784.02	—	—
UC3_auto	131052	-9.53	—	6343.41	-18.51	—
UC3_cg	145373	-0.35	+10.93	703.58	-90.96	-88.91
UC3_pg	257495	+77.75	+96.48	871.32	-88.81	-86.26

achieving 70-80% of saving with respect to both the baseline design and the automatic fine-grain clock gating one. Such a benefit cannot be appreciated in the total power consumption since the static power amount is about the 1-10% of dynamic one. In fact, 90 nm technologies are far from the physical limit where the static power contribution exceeds the dynamic one. A deeper insight on this aspect is provided in Section 4.4.4. In terms of dynamic power consumption MDC power extensions, the clock gating and the power gating ones, allow good saving percentages (more than the 80% with respect to both the baseline and the auto clock gated designs). In comparison with the automatic fine-grain clock gating implemented by SoC Encounter, the coarse-grain approach proposed in this work is capable of getting rid of the clock tree power consumption; therefore, reaches considerably higher saving percentages.

Clock gating does not act on the static power consumption by definition: *UC_cg* designs, due to the added clock gating modules, present a small overhead in terms of static power consumption. *UC_auto* seems to present a very small saving. This latter is due to the area optimizations, visible on Table 4.4, that SoC Encounter performs replacing the standard registers with dedicated clock gating cells.

Focussing on UC2, Figure 4.5 and Figure 4.6 respectively report the static and the dynamic contributes of the kernels involved in the considered UC. In particular, these figures show how the static and the dynamic power consumption of the designs change depending on the enabled kernel. The weighted mean of these data, calculated as described at the beginning of this section, gave the total UC2 numbers reported in Table 4.4 and Table 4.5.

Focusing on the static consumption (reported in Figure 4.5), as expected, power gating is extremely beneficial in reducing the static dissipation for all the kernels. The same does not apply for clock gating as previously discussed. In Figure 4.5 it is possible to notice that one kernel, namely *sbwlabel*, is less positively affected by power gating than the others in terms of static consumption. This is the largest kernel (see Table 4.2) and, when it is active, a considerably high portion of the design is on. Therefore, by construction, it consumes

Table 4.5: 90 nm ASIC synthesis results: focus on static and dynamic power consumption. [* Percentages wrt to the baseline design without power-management; ** Percentages wrt to the baseline design implementing fine-grain clock gating with SoC Encounter]

Design	Static			Dynamic		
	$[\mu W]$	%*	%**	$[\mu W]$	%*	%**
UC1	47.96	—	—	4982.58	—	—
UC1_auto	47.51	-0.93	—	3505.64	-29.64	—
UC1_cg	49.07	+2.32	+3.28	327.02	-93.44	-90.67
UC1_pg	8.73	-81.80	-81.63	481.84	-90.33	-86.26
UC2	57.54	—	—	6124.31	—	—
UC2_auto	56.62	-1.61	—	4939.22	-19.35	—
UC2_cg	58.88	+2.33	+3.99	377.17	-93.84	-92.36
UC2_pg	13.24	-76.99	-76.62	606.45	-90.10	-87.72
UC3	73.92	—	—	7710.10	—	—
UC3_auto	72.39	-2.08	—	6271.03	-18.66	—
UC3_cg	75.27	+1.83	+3.99	628.31	-91.85	-89.98
UC3_pg	15.16	-79.50	-79.06	856.17	-88.90	-86.35

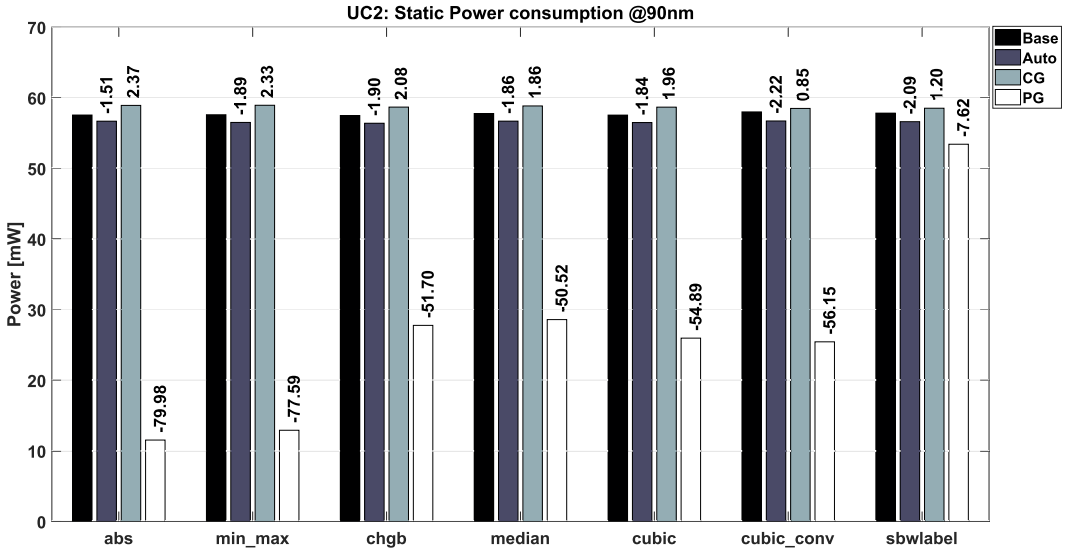


Figure 4.5: UC2: Static power consumption at 90 nm with state retention cells.

more static power than the others. Moreover, the area overhead to manage power gating increases with the state retention cells number, which are potentially more for large kernels. Therefore, according to all these considerations, *sbwlabel* benefits necessarily less than the other kernels from the power gating.

With respect to the dynamic power consumption (reported in Figure 4.6) it is clearly visible that, within the MDC coarse-grain designs, clock gating achieves higher dynamic power savings than power gating, due to the high resource overhead of the latter technique that causes also additional switching activity on the design. The MDC-based designs where power optimization is enabled are capable of reaching higher performance than the register-

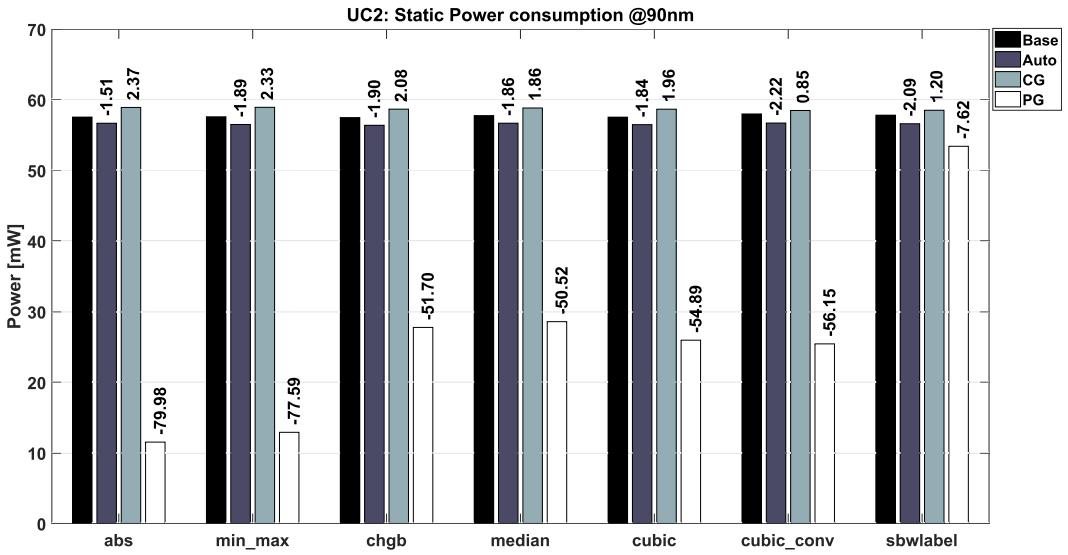


Figure 4.6: UC2: Dynamic power consumption at 90 nm with state retention cells.

level clock gated ones. With respect to the automatic clock gating, whose saving percentages are always within 10%-14%, the MDC-based designs perform better. The best savings are reached for the smaller kernels (see Table 4.2), *abs* and *min_max*, where, by shutting down large portions of the design, coarse-grain clock gating and power gating allow saving more than 91% and 89% respectively. The less positive effect is registered again with *sbwlabel* due to its size (it activates almost the 45% of the entire design).

Summarizing, the coarse-grain clock and power gating approaches, based on the automatic model-based identification of homogeneous logic regions, are capable of achieving better performance than the classical fine-grain clock gating ones (normally implemented in commercial synthesizers), targeting a 90 nm CMOS technology. Strictly focusing on the MDC power extensions, according to the technology adopted in the discussed analysis, clock gating demonstrated to be more efficient than the proposed coarse-grain power gating implementation; despite this, power gating showed very promising results in terms of static power consumption reduction.

4.4.3 90 nm CMOS Technology: application-specific power gating support

In this section, still targeting a 90 nm CMOS technology and the same use-cases, a different set of designs has been assessed. Given multi-functional scenarios such the targeted ones, it is not necessary to maintain the state of the registers when a certain *LR* is switched-off. Results are stored back to the main memory before a new computation is issued on the co-processor, so that all the registers are simply overwritten. According to these considerations, it is not necessary to save the status of the registers prior to switch them off and, in turn, it is possible to get rid of the overhead due to the retention registers in the power gated designs. Results are in line with those discussed on the previous section: MDC-based clock gating and power gating methodologies provide higher performance and all the general considerations still hold.

Focusing on power gating, Table 4.6 summarizes, for each design, the synthesis results in terms of area occupancy and power consumption. The reported data, with respect to Table 4.4, do not include the contribution of the state retention cells. Getting rid of the state retention cells implies a considerably smaller area overhead (7-9% in this second scenario versus 70-90% of the previous one with respect to the baseline designs). Nevertheless, as it can be noticed, the overall power consumption does not change a lot, since with a 90 nm CMOS technology the static power consumption is still far smaller than the dynamic one, which has the largest impact on the power results.

Table 4.6: 90 nm power gating (without state retention cells) ASIC synthesis results. [* Percentages wrt to the baseline design without power-management; ** Percentages wrt to the baseline design implementing fine-grain clock gating with SoC Encounter]

Design	Area			Power		
	$[\mu m^2]$	%*	%**	$[\mu W]$	%*	%**
UC1_pg	99707	+7.06	+17.40	431.57	-91.42	-87.85
UC2_pg	128383	+9.14	+21.14	549.35	-91.11	-89.00
UC3_pg	156230	+7.85	+19.21	769.71	-90.11	-87.87

This section demonstrated that, in those applications where the state preservation is not mandatory, it is possible to potentially benefit from the removal of the state retention cells. Nevertheless, despite the huge area saving, targeting a 90 nm CMOS technology this potential benefit does not turn into a real power saving, since the dynamic power consumption is still orders of magnitude larger than the static one.

4.4.4 Preliminary Results Over a 45 nm Technology

The chosen technology influences the results achievable with the proposed strategies. The 90 nm technology is far from the physical point where the two contributes of the power, static and dynamic, are comparable. With the 90 nm technology the dynamic power is about two orders of magnitude larger than the static one. In such a situation clock gating results always beneficial with respect to power gating. It guarantees large dynamic savings with a negligible area overhead.

This section presents some preliminary results over a smaller technology, targeting 45 nm channel length cells, where the dynamic consumption is approximately just one order of magnitude larger than the static one. All the results reported hereafter refer to the application-specific power gating scenario discussed in Section 4.4.3, without the insertion of the state retention cells. Table 4.7 reports area occupancy and power consumption of the considered designs. The area overhead of the MDC-based coarse-grain solutions is in line to what we have seen in the previous paragraphs for the 90 nm technology: clock gating has a very low overhead (about 0.3% with respect to the baseline design), while power gating is slightly more invasive (it reaches 10% of area overhead for UC3). More interesting numbers are related to the power consumption: coarse-grain power gating designs achieve larger power saving percentages than the corresponding clock gating ones (86-87% of the baseline design power consumption with respect to the 82%). Automatic clock gating behaviour does not change significantly between the two adopted technologies.

Table 4.7: 45 nm ASIC synthesis results. [* Percentages wrt to the baseline design without power-management; ** Percentages wrt to the baseline design implementing fine-grain clock gating with SoC Encounter]

Design	Area			Power		
	$[\mu m^2]$	%*	%**	$[\mu W]$	%*	%**
UC1	25384	—	—	1144.67	—	—
UC1_auto	23360	-7.97	—	804.82	-29.69	—
UC1_cg	25467	+0.33	+9.01	174.07	-84.79	-78.37
UC1_pg	27315	+7.61	+16.93	108.57	-90.52	-86.51
UC2	31564	—	—	1427.69	—	—
UC2_auto	28998	-8.13	—	1163.21	-18.53	—
UC2_cg	31674	+0.35	+9.23	209.39	-85.33	-82.00
UC2_pg	34737	+10.05	+19.79	142.98	-89.99	-87.71
UC3	39040	—	—	1789.00	—	—
UC3_auto	35844	-8.19	—	1456.28	-18.60	—
UC3_cg	39184	-0.37	+9.32	300.08	-83.23	-79.39
UC3_pg	42405	+8.62	+18.30	194.60	-89.12	-86.64

Power gating is more effective than clock gating on a 45 nm technology, despite the two contributes of the overall power consumption still differ for one order of magnitude. As detailed in Table 4.8: power gating presents again better performance in the static power reduction, while clock gating is again more efficient in dynamic power saving. Note that the savings achievable in terms of dynamic power consumption are extremely closed (91-93% for clock gating and 87-88% for power gating). Given this similarity and the fact that only power gating is capable of providing benefits in term of static power consumption (up to 77-82%), the overall power consumption favours power gating solutions (as demonstrated in Table 4.7) for the targeted technology node.

Summarizing, this section confirms that the targeted technology may influence the designer in choosing the more appropriate power saving technique. Considering a technology where the dynamic consumption is the main term of the whole power, clock gating should be the better solution. By adopting a smaller technology, where static and dynamic power are comparable, power gating could guarantee better results.

4.5 Chapter Remarks

This Chapter addressed the problem of power management in CGR systems. Such systems are as suitable to accelerate multi-functional applications as, potentially, energy inefficient. In fact, on a CGR substrate, while a particular task is executed, the resources not involved in the computation may potentially waste precious power if not properly managed. Therefore, a power management methodology that extends the already present in MDC has been presented. In particular, the proposed methodology extends the capability of MDC of identifying disjointed homogeneous logic regions to include also the combinational logic. These regions have been successfully adopted to implement a coarse grained power gating based power saving technique. Together with the HDL multi-functional design, MDC provides a

Table 4.8: 45 nm ASIC synthesis results: focus on static and dynamic power consumption. [* Percentages wrt to the baseline design without power-management; ** Percentages wrt to the baseline design implementing fine-grain clock gating with SoC Encounter]

Design	Static			Dynamic		
	[μW]	%*	%**	[μW]	%*	%**
UC1	102.86	—	—	1041.81	—	—
UC1_auto	102.05	-0.79	—	702.77	-32.54	—
UC1_cg	105.59	+2.65	+3.47	68.48	-93.43	-90.26
UC1_pg	18.35	-82.16	-82.02	90.22	-91.34	-87.16
UC2	126.54	—	—	1301.15	—	—
UC2_auto	121.41	-4.05	—	1041.79	-19.93	—
UC2_cg	129.85	+2.61	+6.95	79.54	-93.83	-92.36
UC2_pg	28.18	-77.73	-78.30	114.80	-91.18	-88.98
UC3	158.99	—	—	1630.02	—	—
UC3_auto	154.08	-3.09	—	1302.21	-20.11	—
UC3_cg	163.10	+2.59	+5.86	136.97	-91.60	-89.48
UC3_pg	30.47	-80.83	-81.32	164.13	-89.93	-87.40

common power format file (CPF) to give to the synthesizer the information about the power intent.

The potential of the power manager extension have been proven on different image processing application scenarios. MDC power extensions are capable of providing both coarse-grain clock gating and power gating support. For multi-functional reconfigurable system, it has been demonstrated that: 1) the automatic fine-grain clock gating techniques (as those available on commercial synthesizers) are not sufficient for power saving purposes; 2) according to the adopted technology, the applications characteristics and the constraints to be met (area/power/frequency), it is possible to opt for one of the two supported techniques to achieve optimal performances.

Results assessment, on ASIC, demonstrated that the dynamic power manager can lead to the 90% of power saving, in highly variable multi-functional scenarios. However, depending on the technology power gating is not always the best strategy. Thus, a smarter power saving strategy that combines the coarse-grain power gating and clock gating strategies, to selectively decide which one between clock and power gating better suites to the considered logic region is going to be discussed in Chapter 5.

List of Publications Related to the Chapter

Journal papers

- Francesca Palumbo, Tiziana FANNI, Carlo Sau, and Paolo Meloni. 2017. *Power-Awarness in Coarse-Grained Reconfigurable Multi-Functional Architectures: a Dataflow Based Strategy.* J. Signal Process. Syst. 87, 1 (April 2017), 81-106. DOI: https://doi.org/10.1007/s11265-016-1106-9

Conference papers

- Tiziana FANNI, Carlo Sau, Luigi Raffo, and Francesca Palumbo. *Automated power gating methodology for dataflow-based reconfigurable systems.* In Proceedings of the 12th ACM International Conference on Computing Frontiers (CF '15), 2015. ACM, New York, Article 61 , 6 pages. DOI: http://dx.doi.org/10.1145/2742854.2747285

- Subhadeep Banik, Andrey Bogdanov, Tiziana FANNI, Carlo Sau, Luigi Raffo, Francesca Palumbo, and Francesco Regazzoni. 2016. Adaptable AES implementation with power-gating support. In Proceedings of the ACM International Conference on Computing Frontiers (CF '16). ACM, New York, NY, USA, 331-334. DOI: https://doi.org/10.1145/2903150.2903488

Other scientific papers

- Francesca Palumbo, Carlo Sau, Tiziana FANNI and Luigi Raffo, *Reconfigurable Platform Composer Tool Project*, at the 2016 Riunione Annuale del Gruppo Elettronica (GE), Brescia (Italy) June 2016.

Chapter 5

Coarse-Grain Reconfiguration on ASIC - A Power Model and Optimization Procedure for Mixed Clock-Gating and Power-Gating

Coarse-grain reconfigurable (CGR) systems are suitable to accelerate multi-functional applications. In the research of this thesis a dataflow-based approach has been exploited to manage the composition of CGR systems with the automated implementation of power gated platforms, where the idle logic can be powered down. However, the power gating is a power saving technique that requires several additional logic, and the resource overhead may easily overcome the power saved by switching off the unused logic. The main contribution of this chapter is a power modelling methodology capable of determining, prior to any physical system implementation, which is the optimal power saving strategy for a CGR system. This methodology is integrated in the automated flow of MDC power management extension. The work presented in this Chapter has been conducted in a collaboration between the Microelectronics and Bioengineering Lab (EOLAB) (University of Cagliari) and the Intelligent system DEsign and Applications (IDEA) Lab (University of Sassari), into the context of the "RPCT - Reconfigurable Platform Composer Tool" Project, funded by the Sardinian Regional Government under agreement *L.R. 7/2007, CRP-18324*.

5.1 State of the Art: Modelling Power Consumption in Coarse-Grain Reconfigurable ASIC architectures

The power gating is a power saving technique that can be extremely beneficial in saving both static and dynamic power. However, as seen in Chapter 4, power gating is a quite invasive technique that requires several additional logic, and blindly shutting-off the idle logic is not always the best strategy. In some cases the power consumption due to the power saving logic might exceed the amount of power saved by switching-off the idle logic — this can happen for small idle regions. In other cases, the power gating is not as efficient as the much simpler clock gating — for instance in regions that have mainly sequential logic. For these reason it could be useful analysing the design to identify

which portions of the design can benefits of power saving techniques.

When the architecture is manually designed and implemented and only few idle areas are involved, such an analysis is affordable. However, talking about CGR systems, there could be tens of regions that are idle during the execution of one or more configurations. In such a case, identifying the best power saving strategy could be highly time consuming. Several works dealt with the issue of modelling power consumption in digital circuits, with the goal of designing low-power circuits. Helms et al. [43] presented an efficient and accurate analytical per gate leakage model that takes all relevant leakage effects into account. It can predict leakage currents without needing any SPICE simulations for model application. Paim et al. [79] presented a power-predictive environment for power-aware FIR filter design based on Remez algorithm. The used technique searches for the optimal combination of number of taps and bit-width for the power-aware ASIC based FIR filter design. The proposed power-predictive environment enabled a fast and power-aware decision even in mathematical design level, reducing the power dissipation and the time-to-market of the chip. Nasser et al. [74] propose a power estimation methodology to allow the designer to explore various hardware architectures in terms of power consumption and performances. These power models are based on neural networks that predict the power consumed by digital operators or IPs like the arithmetic operators (adders, multipliers and memories), implemented on FPGA. However none of them consider any power saving strategy.

Other approaches perform an estimation that considers different components. Li et al. [60] proposed an architecture-level integrated power, area, and timing modelling framework for multicore systems, that evaluates system building blocks (i.e., CPU, buses, etc.) for different technology nodes, providing also power gating support. The CASPER simulator for shared memory many-core processors [29] includes pre-characterized libraries containing power dissipation models of different hardware components, enabling accurate power estimation at a high-level exploration stage. In particular the authors implemented Chipwide Dynamic Voltage, Frequency Scaling, and Performance Aware Core-Specific Frequency Scaling. The FALPEM framework [23] provides power estimations at pre-register transfer level (RTL) stage, specifically targeting the power consumed by clock network and interconnect, but power and clock gating costs were not defined. Stokke at al. [111] proposed a power modelling method for the Tegra K1 CPU, that taking into account measured rail voltages and fine-grained hardware activity predictors, expose components such as rail and core leakage currents. Finally, the works in [134, 73] focused on networks-on-chip. Zoni et al. [134] proposed a cycle-accurate simulation framework to support exploration and optimization of the power and performance metrics during network-on-chip design. It provides accurate models for both DVFS and power gating actuators, encompassing their power and performance overheads. Such overheads are integrated and added into the timing and power consumption figures of the architecturally simulated components. Nasirian et al. [73] presented and approach for power gating management in network-on-chip designs. They modelled the behaviour of router buffers using queuing theory to evaluate the effect of power gating on the overall power saving and power penalty. To evaluate the proposed algorithm on network-on-chip system, they adopted a cycle accurate simulator.

However, no works were found to address both clock gating and power gating estimation in CGR systems. Some approaches only partially address the issue. For example, Xu et al. [125] derived two energy models for estimating the leakage reduction for power gating and reverse body bias. These models could accurately estimate the circuit energy saving at any time, even when the circuit is in state transition. Shafique et al. [101] focussed on low-power techniques and power modelling for FPGAs. They presented a runtime adaptive energy management system able of dynamically determining a set of energy-minimizing custom instructions implementation versions under the given performance and area constraints. Then it power gates the idle subset of the custom instructions, switching off the data path containers of the fabric. In [128], only clock gating is taken into account: authors proposed a high-level power model based on clock gating enable signals whose different power states were defined by the combination of the values of clock gating enable signals. Their consumption

was characterized by low-level power analysis results. Shyu et al. [104] proposes a methodology to construct a hybrid routing structure to connect power switches that alleviates rush current without violating the wake-up sequence time constraint. To determine depth of a daisy chain, they propose a simplified model for power gating design, to estimate voltage and transient current. However the model is not used to determine whether it is convenient to switch off the idle logic.

With respect to above considered works, this Chapter present power modelling methodology and an improvement of the MDC power manager by integrating such methodology within its automated flow. This Chapter introduces an algorithm that analyses the identified logic regions and, on the basis of one single synthesis and a minimal set of simulations (one for each scenario of the multi-functional problem), is capable of optimally characterizing the power management support. This flow, in a separate manner for each logic region of the CGR design, is capable of assessing both clock and power gating management costs and of determining which is the optimal power saving strategy (if any) prior to any physical system implementation. The algorithm is based on detailed static and dynamic power consumption models that take into account functional, architectural and technological parameters to define the potential overhead and benefits of the considered solutions.

5.2 Methodology

To overcome the limits of a blindly applied unique power management strategy, this section presents a power estimation flow capable of characterising, at a high-level of abstraction, the *LRs* identified by the MDC power extension, and to estimate power and clock gating overhead before any physical implementation. The estimation is based on two sets of models that determine the static and dynamic consumptions of each *LR* when clock gating or power gating are applied. The proposed models are derived after a single logic synthesis of the baseline CGR system generated by MDC, carried out with commercial synthesis tools from the analysis of the power reports obtained after netlist simulation.

Given any hardware Functional Unit (FU) its average power can be obtained by summing up the single contributions of the adopted cells, provided by the targeted ASIC library. In particular, the average power consumption of an FU (Equation 5.1), can be divided into the power consumed by its combinatorial logic ($P(cmb)$) and the power consumed by its sequential logic ($P(reg)$).

$$P(FU) = P(cmb) + P(reg) \tag{5.1}$$

And the power consumption of an *LR* can be obtained by summing up the contributions of its FUs, uniquely corresponding to the actor of the reconfigurable IR (Equation 5.2). Equation 5.2 is valid for both static and dynamic power. However, the static power consumption is tightly related to the *LR* area: the more cells are included in the considered region, the more is its corresponding static dissipation. The dynamic power consumption strongly depends on the nodes switching activity.

$$P(LR_i) = \sum_{actors \in LR_i} [P_i(cmb) + P_i(reg)] \tag{5.2}$$

Given these considerations, the power consumption of an *LR* when the clock gating or power gating are applied is given by (1) the baseline power consumption of the *LR* (from Equation 5.2) weighted by the activation factor of the *LR* ($T_{i_{ON}}$) plus (2) the power consumption of the logic inserted into the *LR* to apply the power saving technique. Following sections present the power estimation models for a given *LR* when power gating is applied (Section 5.2.1) and when clock gating is applied (Section 5.2.2).

5.2.1 Power Gating - Power Consumption Models

Equation 5.3 models the static power consumption (due to leakage currents, so in this chapter also called *leakage power* or simply P_{lkg}) of a *LR* with a prospective power gating implementation. It

involves two terms: $P_{lkgON}(LR_i)$ corresponds to the static consumption within the considered *LR* and $Ext\_Over_{lkg}(LR_i)$ refers to the power overhead due to the power gating logic inserted outside the *LR*. This second term does not consider the *Power Switch* overhead, since it is not included in the pre-layout netlist. Power gating prevents, by definition, any static dissipation on the *LR* when disabled; therefore, Equation 5.3 does not present any $P_{lkgOFF}(LR_i)$ related to the *LR*.

$$\begin{aligned} &P_{lkg}(LR_i) = P_{lkgON}(LR_i) + Ext\_Over_{lkg}(LR_i) = \\ &= \sum_{actors \in LR_i} [P_{lkg}(cmb) + P_{lkg}(reg) * (\#reg - \#rtn)/\#reg + P_{lkg}(RC) * \#rtn] * T_{i_{ON}} + \\ &+ [P_{lkg}(ISO_{ON}) * T_{i_{ON}} + P_{lkg}(ISO_{OFF}) * T_{i_{OFF}}] * \#iso + \\ &+ [P_{lkg}(Contr_{ON}) * T_{i_{ON}} + P_{lkg}(Contr_{OFF}) * T_{i_{OFF}}] + \\ &+ [P_{lkg}(CG_{ON}) * T_{i_{ON}} + P_{lkg}(CG_{OFF}) * T_{i_{OFF}}] \end{aligned} \tag{5.3}$$

$P_{lkgON}(LR_i)$ is obtained as the multiplication of the *LR* activation factor $T_{i_{ON}}$ and the sum of static power of the involved actors, considering separately combinatorial and sequential logic. The former, $P_{lkg}(cmb)$, is equal to the static of the combinatorial cells within the considered *LR*. The sequential logic part is related to the number of registers (#reg) within the *LR*, and involves two terms: (1) the first term refers to the registers whose state can be lost and it is estimated on the basis of the static consumption of the sequential cells ($P_{lkg}(reg)$), as an average on the number of registers that are not retained; (2) the second one refers to the retention cells and it is estimated starting from the number of registers whose state has to be maintained ($\#rtn$) multiplied by the static power of a single *State Retention* cell ($P_{lkg}(RC)$), which value is retrieved from the target ASIC library.

$Ext\_Over_{lkg}(LR_i)$ is composed of three terms: the first one is related to the *Isolation* cells ($\#iso$), the second one to the *Power Controller* and the third one to the clock gating cell[1]. Note that, unlike P_{lkgON}, for the three above-mentioned terms Ext_Over_{lkg} characterizes the *LR* static consumption in both its on and off states. In the on state, the model accounts for the static consumption in the on state (e.g. $P_{lkg}(ISO_{ON})$) multiplied by the activation factor $T_{i_{ON}}$ and by the overall number of cells within the *LR* (e.g. $\#iso$). In the off state, the model accounts for the static consumption in the off state (e.g. $P_{lkg}(ISO_{OFF})$) multiplied by the inactivation factor $T_{i_{OFF}}$ (given by $1 - T_{i_{ON}}$) and by the overall number of cells within the *LR* (e.g. $\#iso$). Please note that there is just one *Power Controller* for all the *LRs* and one clock gating cell per *LR*, but an a-priori characterisation phase is required to the designer, since their consumption values cannot be retrieved directly from any ASIC library.

Frequently, commercial tools (e.g. Cadence Encounter Digital Implementation System) consider dynamic power as composed of two main terms, as depicted by Equation 5.4: a *net* contribution due to the power dissipated throughout the wires linking the cells, and an *internal* contribution due to the dissipation occurring inside the cells [16]. The operating frequency, f, influences both terms. P_{net} accounts for the load capacitance of each net_j (bearing a specific capacitance C_{load_j}) and the related switching activity (SW_j). Whereas, P_{int} depends on the power per MHz dissipated by each cell (P_i) and the related switching activity (SW_i).

$$\begin{aligned} P_{dyn} &= P_{net} + P_{int} \\ &= \frac{1}{2} f V_{DD}^2 \sum_{net_j} C_{load_j} SW_j + f \sum_{cell_i} P_i SW_i \end{aligned} \tag{5.4}$$

Estimation model for dynamic power consumption described by Equation 5.5 reflects the model for the static power consumption (Equation 5.3). The main difference among the static power model

[1] The power gating switch off protocol requires to apply clock gating at the region level, before retaining the registers value.

and the dynamic one is that this latter requires accurate data in terms of nodes switching activity. For this reason, the netlist of the baseline CGR system is not sufficient to retrieve accurate values from the power reports and one different simulation of the netlist for every implemented functionality is required. Thus, dynamic power model takes into consideration the real system switching activity provided by the hardware simulations.

$$\begin{aligned} &P_{int}(LR_i) = P_{intON}(LR_i) + Ext\_Over_{int}(LR_i) = \\ &= \sum_{actors \in LR_i} [P_{int}(cmb) + P_{int}(reg) * (\#reg - \#rtn)/\#reg + P_{int}(RC) * \#rtn] * T_{i_{ON}} + \\ &+ [P_{int}(ISO_{ON}) * T_{i_{ON}} + P_{int}(ISO_{OFF}) * T_{i_{OFF}}] * \#iso + \\ &+ [P_{int}(Contr_{ON}) * T_{i_{ON}} + P_{int}(Contr_{OFF}) * T_{i_{OFF}}] + \\ &+ [P_{int}(CG_{ON}) * T_{i_{ON}} + P_{int}(CG_{OFF}) * T_{i_{OFF}})] \end{aligned} \tag{5.5}$$

The current model for dynamic power consumption considers only the P_{int} contribution that can be expressed for each single *LR*, the P_{net} term of Equation 5.4 is not currently addressed in the model. Nevertheless, as will be demonstrated in Section 5.4 (please see Tables 5.9,5.8, 5.12 and 5.14) neglecting this term seems not to affect the optimal identification of the region to be gated.

5.2.2 Clock Gating - Power Consumption Models

Clock gating static and dynamic models are less complicated than the power gating ones, since clock gating requires a very low logic overhead and it positively acts only on the dynamic dissipation. Equation 5.6 and Equation 5.7 report the models adopted respectively for the static power estimation and for the dynamic power one, referring to a clock gated design.

$$\begin{aligned} &P_{lkg}(LR_i) = P_{lkg}(LR_i) + Ext\_Over_{lkg}(LR_i) = \\ &= \sum_{actors \in LR_i} [P_{lkg}(cmb) + P_{lkg}(reg)] + \\ &+ [P_{lkg}(Enab_{ON}) * T_{i_{ON}} + P_{lkg}(Enab_{OFF}) * T_{i_{OFF}}] + \\ &+ [P_{lkg}(CG_{ON}) * T_{i_{ON}} + P_{lkg}(CG_{OFF}) * T_{i_{OFF}}] \end{aligned} \tag{5.6}$$

$$\begin{aligned} &P_{int}(LR_i) = P_{int}(LR_i) + Ext\_Over_{int}(LR_i) = \\ &= P_{int}(combLR_i) + P_{intON}(seqLR_i) + Ext\_Over_{int}(LR_i) = \\ &= \sum_{actors \in LR_i} [P_{int}(cmb) + P_{int}(reg) * T_{i_{ON}}] + \\ &+ [P_{int}(Enab_{ON}) * T_{i_{ON}} + P_{int}(Enab_{OFF}) * T_{i_{OFF}}] + \\ &+ [P_{int}(CG_{ON}) * T_{i_{ON}} + P_{int}(CG_{OFF}) * T_{i_{OFF}})] \end{aligned} \tag{5.7}$$

At the logic region level, Equation 5.6 considers always the combinatorial and sequential contributions for both the ON or OFF states clock gating does not affect the system static power. Whereas, Equation 5.7 considers always the combinatorial part for both the ON or OFF states (combinatorial logic cannot benefit from clock gating) and the sequential contribution only during the *LR* active time. The overhead, $Ext\_Over_{int}(LR_i)$, is given by the clock gating cell and the *Enable Generator*. Please remember that, implementing clock gating management at a coarse-grain level, just one clock gating cell per *LR* has to be inserted within the system. Equation 5.7 is pretty much the same as Equation 5.6, but dealing with the dynamic model, clock gating effects are estimated by omitting the contribute of sequential logic when the *LR* is OFF.

5.2.3 Parameters Discussion

The proposed models are determined by the intrinsic features of the *LRs*. In particular, they consider:

- *architectural parameters*: *LRs* composition determines the amount of involved combinatorial and sequential cells;
- *functional parameters*: *LRs* behaviour defines the region activation factor and if its status has to be preserved or not;
- *technological parameters*: target technology has an impact on the ratio between dynamic and static power and on the different cells characterisation.

Table 5.1 reports, for each parameter considered in equations 5.3, 5.5, 5.6 and 5.7, their classification. A deeper explanation about $P_{lkg/int}(cmb)$ and $P_{lkg/int}(reg)$ is necessary. They are not associated with any specific parameters class, indeed they depend on type and number of involved cells composing the considered *LR* and also on the system switching activity (especially for the Internal contribute). These values are gathered by the reports of the baseline (without any application of power saving techniques) CGR system netlist, assuming that the amount and type of cells composing the FUs do not change as power saving strategies are applied (except for the retained registers). Since technology parameters are the result of power reports, the power estimation equations are valid and accurate for different technology without any adaptation in the formulas (as it will be demonstrated in Section 5.4).

Table 5.1: Parameters classification. Table depicts for each parameters the typologies it belong (*architectural*, *functional* and *technological*), its description and and how it is exacted.

Parameter	arch.	funct.	tech.	Description	Extraction
$\#reg$	x			Number of sequential cells in the considered *LR*	Provided by the synthesis reports
$\#iso$	x			Number of estimated isolation cells in the design	Obtained by counting the number of wires that connect the different *LR* among each other in the dataflow model.
$T_{i_{ON}}$ $T_{i_{OFF}}$		x x		Activation Factor OFF Time	Found by means of a high-level (directly on the dataflow) profiling of the targeted scenario
$\#rtn$		x		Number of retention cells in the design	It is strictly related to the *LR* functionality and is chosen by the designer.
$P_{lkg/int}(RC)$ $P_{lkg/int}(ISO_{ON})$ $P_{lkg/int}(ISO_{OFF})$			x x x	Power Estimation of retention and isolation cells.	Determined by the selected synthesis process and can be obtained without any implementation run.
$P_{lkg/int}(Contr_{ON})$ $P_{lkg/int}(Contr_{OFF})$ $P_{lkg/int}(Enab_{ON})$ $P_{lkg/int}(Enab_{OFF})$			x x x x	Power Estimation of PG and CG controller when the *LR* is ON or OFF.	It requires to be characterised, according to the target technology, with dedicated synthesis trials.
$P_{lkg/int}(cmb)$ $P_{lkg/int}(reg)$	x x	x x	x x	Power consumed by combinatorial and sequential cells within the considered *LR*.	Gathered by the reports of the baseline CGR system netlist.

5.2.4 Power Analysis Algorithm

The presented power estimation models are exploited into Algorithm 4 (see Appendix C) that guides the designers towards the optimal solution for each *LR* (if apply clock gating, power gating or not apply any power saving technique at all), rather than choosing a one-fit-to-all switching-OFF technique for all of them. For each *LR*, Algorithm 4 executes the following steps, embodied by different functions:

1. *area thresholding* (see *evaluate_area* function in Algorithm 4): power gating is a quite invasive technique, requiring several additional logic in the non-switchable always-ON domain. Thus, it is possible to assume that power gating cannot bring any benefit for for small *LRs*. Then *LRs* under a certain area threshold, defined by the user, are not considered for implementation. However, clock gating may still be beneficial, due to its very small additional logic amount.

2. *power gating evaluation* (see *evaluate_PG* function in Algorithm 4): power gating cost is estimated in order to find out if it can lead to power saving or not. The prospective power and clock gating implementations are compared on the basis of their overall consumption. Equation 5.3 is applied and summed to Equation 5.5, if there is not total power saving the algorithm goes to the *clock gating variation estimation*. On the contrary, if there is saving it has to be compared with the sum of Equation 5.6 and Equation 5.7 to determine whether the current *LR* may benefit from power gating (despite its larger overhead) or from clock gating.

3. *clock gating evaluation* (see *evaluate_CG* function in Algorithm 4): clock gating cost is estimated to investigate the possibility of achieving power saving with this technique. If the achievable saving does not counterbalance clock gating implementation costs, the *LR* is discarded. This means that the *LR* logic is included in the always-ON domain. On the contrary, if the clock gating leads to an overall saving in terms of total power, the *LR* can be candidate for clock gated implementation.

5.3 Integration in MDC

The discussed estimation models (described in Equation 5.3, Equation 5.5, Equation 5.6 and Equation 5.7), together with the Algorithm 4, have been integrated in the MDC design flow, to implement a fully automated power management strategy. Figure 5.1 provides an overview of the modified MDC design flow, MDC tool and its power management extension are directly interfaced with the logic synthesizer and the Algorithm 4 is implemented within the *Power Analysis* block. MDC baseline tool provides the HDL description of the plain CGR system and all the scripts to perform the synthesis of the CGR design and all the different hardware simulations (one for each input DPN), as required by the proposed power estimation models. The power reports are then fed back to the MDC power management extension and parsed within the *Power Analysis* to execute Algorithm 4. In particular, to identify the best power saving strategy for the current *LR*, under evaluation in the Algorithm 4, its power consumption for clock or power gating application is automatically estimated using the data from Table 5.1 that are (1) gathered by provided power reports (as $\#reg$, $P_{lkg/int}(cmb)$, $P_{lkg/int}(reg)$...) and (2) provided by MDC (as the *LRs* themselves and the $\#iso$) (3) result of characterisation (as $P_{lkg/int}(Contr_{ON})$ and the rest of the control logic). The *LRs* classification (see $LRclass$ in Figure 5.1), generated by the *Power Analysis* block, is used by the *CG/PG HDL Generation* block to automatically define and implement the hybrid, clock and power gating, power management support for the given CGR design.

This flow does not require designers to opt for a specific power management technique. On the basis of the presented power estimation models and by linking MDC with a logic synthesis tool, it

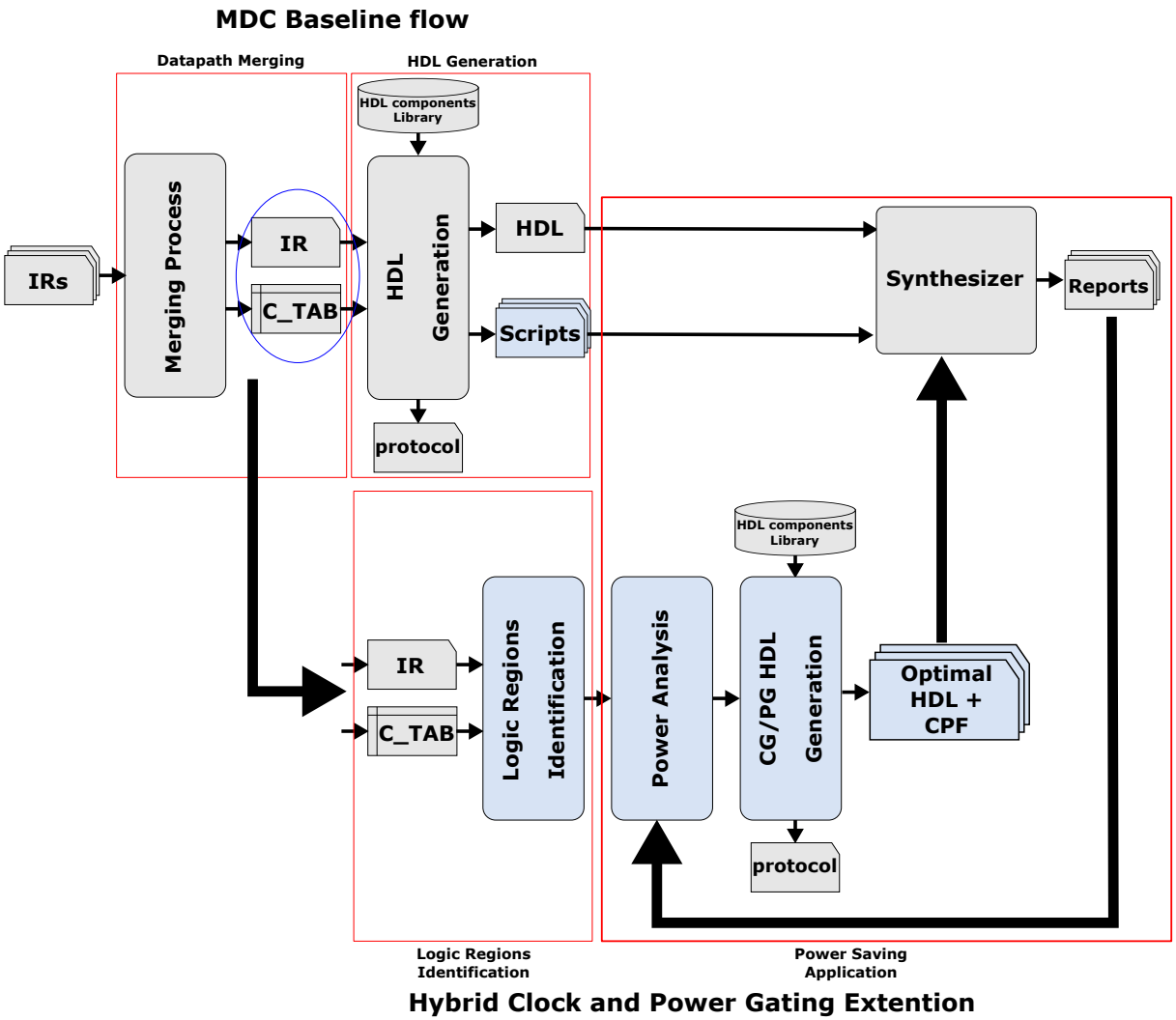


Figure 5.1: Enhanced MDC design suite: integration of the automated hybrid, clock and power gated, support.

is capable to overcome the limit of providing a one-fit-to-all solution. Each *LR*, in a CGR design, is supported (where necessary) with the optimal power management technique.

5.3.1 Step-by-step example

To better explain the presented methodology, this Section considers the example proposed in Figure 4.1, where three input DPNs are merged in a CGR system and five *LRs* have been identified. Since *LR2* is activated by all of the DPNs, this region is always on and it is automatically discarded by the power analysis.

Table 5.2 depicts the composition of each *LRs* in terms of actors and low power logic. It depicts also, for each *LRs* which kernels activate it, and which is the related activation factor T_{ON}. The power consumption values of each actor, have been extracted by the synthesis reports of the baseline (with no power saving applied) CGR platform and required three hardware simulations (one for each input kernel). These simulations are necessary to correctly estimate the internal power consumption of the different *LRs*, taking into account the real switching activity of the design. In practice, power values are determined as an average of those obtained according to the different switching activity profiles.

Table 5.2: Parameter and power consumption of each *LR*, extracted by the synthesis reports of the baseline CGR platform.

Logic Region	kernel	T_{ON}	actors	$\#iso$	$\#reg$	$\#rtn$
LR1	α	0.1	B	32	514	24
LR3	β	0.6	D, E	32	8	8
LR4	α, γ	0.4	A, *SB_0*, SB_1	96	256	64
LR5	γ	0.3	F, G	32	265	192
Actor	Power [nW]					
	lkg seq.	int seq	lkg comb.	int comb.	$\#reg$	$\#rtn$
B	801	104987	121411	3916599	512	24
D	48	1104	51	319	4	4
E	56	1437	53	198	4	4
A	3264	89238	0	0	256	64
SB_0	0	0	307	409	–	–
SB_1	0	0	225	350	–	–
F	1232	22489	213	537	128	128
G	1385	44068	273	363	128	64

Table 5.3 depicts the static and internal power consumption of the additional low power logic. As defined in Section 5.2.3, the parameters reported in Table 5.3 are extracted by the reference technology library or characterized by synthesis trials (see the definition provided in Table 5.1).

Table 5.3: Contributions of static and internal power consumption extracted by the reference technology library or characterized by synthesis trials.

Parameters	lkg power [nW]	int power [nW]
$Enab_{ON}$	84.51	1351
$Enab_{OFF}$	76.54	1320
$Contr_{ON}$	95.44	1449
$Contr_{OFF}$	88.63	1488
CG_{ON}	5.77	169
CG_{OFF}	4.71	292
Iso_{ON}	4.27	2.7
Iso_{OFF}	1.39	0
$P(RC)$	17.15	383.25

Starting from the data in Table 5.2 and Table 5.3, here follows the detailed equations characterization for *LR5*, which includes actors *F* and *G*. When power gating is applied the static power consumption of *LR5* is derived according Equation 5.3, as follows:

$$
\begin{aligned}
&P_{lkg}(LR5) = \\
&= [(P_{lkg}(comb_F) + P_{lkg}(RC) * \#rtnF + P_{lkg}(reg_F) * (\#regF - \#rtnF)/\#regF) + \\
&+ (P_{lkg}(comb_G) + P_{lkg}(RC) * \#rtnG + P_{lkg}(reg_G) * (\#regG - \#rtnG)/\#regG)] * T5_{ON} + \\
&+ [P_{lkg}(ISO_{ON}) * T5_{ON} + P_{lkg}(ISO_{OFF}) * T5_{OFF}] * \#iso_5 + \\
&+ [P_{lkg}(Contr_{ON}) * T5_{ON} + P_{lkg}(Contr_{OFF}) * T5_{OFF}] + \\
&+ [P_{lkg}(CG_{ON}) * T5_{ON} + P_{lkg}(CG_{OFF}) * T5_{OFF}] = \\
&= [(213 + 17.15 * 128) + \\
&+ (273 + 17.15 * 64 + 1385 * 0.5)] * 0.3 +
\end{aligned}
$$

$+[4.27 * 0.3 + 1.39 * 0.7] * 32+$
$+[95.44 * 0.3 + 88.63 * 0.7]+$
$+[5.77 * 0.3 + 4.71 * 0.7] = 1509.219$

The internal power consumption is given by Equation 5.5:
$P_{int}(LR5) =$
$= [(P_{int}(comb_F) + P_{int}(RC) * \#rtnF + P_{int}(reg_F) * (\#regF - \#rtnF)/\#regF)+$
$+(P_{int}(comb_G) + P_{int}(RC) * \#rtnG + P_{int}(reg_G) * (\#regG - \#rtnG)/\#regG)] * T5_{ON}+$
$+[P_{int}(ISO_{ON}) * T5_{ON} + P_{int}(ISO_{OFF}) * T5_{OFF}] * \#iso_5+$
$+[P_{int}(Contr_{ON}) * T5_{ON} + P_{int}(Contr_{OFF}) * T5_{OFF}]+$
$+[P_{int}(CG_{ON}) * T5_{ON} + P_{int}(CG_{OFF}) * T5_{OFF})] =$
$= [(537 + 383.25 * 128)+$
$+(363 + 383.25 * 64 + 44068 * 0.5)] * 0.3+$
$+[2.7 * 0.3 + 0 * 0.7] * 32+$
$+[1449 * 0.3 + 1488 * 0.7]+$
$+[169 * 0.3 + 292 * 0.7] = 30217.72$

When clock gating is considered Equation 5.6 and Equation 5.7 are computed as follows:
$P_{lkg}(LR5) =$
$= [(P_{lkg}(comb_F) + P_{lkg}(reg_F)) + (P_{lkg}(comb_G) + P_{lkg}(reg_G))]+$
$+[P_{lkg}(Enab_{ON}) * T5_{ON} + P_{lkg}(Enab_{OFF}) * T5_{OFF}]+$
$+[P_{lkg}(CG_{ON}) * T5_{ON} + P_{lkg}(CG_{OFF}) * T5_{OFF}] =$
$= [(213 + 1232) + (273 + 1385)]+$
$+[84.51 * 0.3 + 76.54 * 0.7]+$
$+[5.77 * 0.3 + 4.71 * 0.7] = 3186.96$

$P_{int}(LR5) =$
$= [(P_{int}(comb_F) + P_{int}(reg_F) * T5_{ON}) + (P_{int}(comb_G) + P_{int}(reg_G) * T5_{ON})]+$
$+[P_{int}(Enab_{ON}) * T5_{ON} + P_{int}(Enab_{OFF}) * T5_{OFF}]+$
$+[P_{int}(CG_{ON}) * T5_{ON} + P_{int}(CG_{OFF}) * T5_{OFF})] =$
$= [(537 + 22489 * 0.3) + (363 + 44068 * 0.3)]+$
$+[1351 * 0.3 + 1320 * 0.7]+$
$+[169 * 0.3 + 292 * 0.7] = 22451.8$

Table 5.4: Resulting power consumption of the different *LRs* when the proposed models are applied

Logic Region	lkg PG [nW]	int PG [nW]	lkg CG [nW]	int CG [nW]
LR1	1240.69	404358.71	122294.15	3928700.60
LR3	342.67	3884.44	294.67	3599
LR4	2062.11	38705.08	3880.86	5598.6
LR5	1509.58	30712.72	3186.96	22451.8

Table 5.4 summarizes all the values achieved applying the proposed static and dynamic models to all the different logic regions. These values, compared with consumption of baseline (not-gated) *LRs*, are used into Algorithm 4 to calculate the power variation of each *LR* due to clock gating and power gating strategies. Figure 5.2 explicates the application of the Algorithm 4 when threshold on the area ($area_{th}$) is set to 5%.

- LR_1 is processed by invoking *evaluate_area(LR_1,5)*.
  - Its area is calculated: $area_{LR1}$ =52% of total area.
  - $area_{LR1} > area_{th}$, so that a prospective power gating implementation on LR_1 is taken into consideration by invoking *evaluate_PG(LR_1)*.
    - The static and the dynamic variations are estimated, respectively applying Equation 5.3 and Equation 5.5. The power gating variation on the overall consumption is calculated by subtracting the power consumption of the *LR* when PG is applied, to the power consumption of the *LR* in the baseline design, the result is then divided by the total power consumption of the baseline design in order to estimate the total percentage power variation. The power variation when PG is applied to region LR_1 is -86.45%. Since this value is negative, power gating may be convenient if its total saving is larger than the clock gating one.
    - Equation 5.6 is calculated and summed up to Equation 5.7 to determine clock gating variation on the overall consumption, which is -2.15%.
    - Power gating is more beneficial than clock gating determining, overall, a larger power saving. Thus LR_1 is added to *PG_set*.
- LR_3 is processed by invoking *evaluate_area(LR_3,5)*.
  - Its area is calculated: $area_{LR3}$ =0.4% of total area.
  - $area_{LR3} < area_{th}$, so that a prospective clock gating implementation on LR_3 is considered straight away by invoking *evaluate_CG(LR_3)*.
    - Equation 5.6 and Equation 5.7 are evaluated to determine clock gating variation on the overall consumption: CG_{over} =+0.01%
    - Clock gating is not beneficial since its variation is positive. Thus LR_3 is discarded and no power management policy will be applied to it.
- LR_4 is processed by invoking *evaluate_area(LR_4,5)*.
  - Its area is calculated: $area_{LR4}$ =7% of total area.
  - $area_{LR4} > area_{th}$, so that a prospective power gating implementation on LR_4 is taken into consideration by invoking *evaluate_PG(LR_4)*.
    - The static and the dynamic variations are estimated, respectively applying Equation 5.3 and Equation 5.5. The power gating variation on the overall consumption is -1.2%. Since this value is negative, power gating may be convenient if its total saving is larger than the clock gating one.
    - Equation 5.6 is calculated and summed up to Equation 5.7 to determine clock gating variation on the overall consumption, which is -2.0%.
    - Clock gating is more beneficial than power gating determining, overall, a larger power saving. Thus LR_4 is added to *CG_set*.
- Finally, LR_5 is processed by invoking *evaluate_area(LR_5,5)*.
  - Its area is calculated: $area_{LR5}$ =15% of total area.
  - $area_{LR5} > area_{th}$, so that a prospective power gating implementation on LR_5 is taken into consideration by invoking *evaluate_PG(LR_5)*.

* The static and the dynamic variations are estimated, respectively applying Equation 5.3 and Equation 5.5. The power gating variation on the overall consumption is -0.89%. Since this value is negative, power gating may be convenient if its total saving is larger than the clock gating one.
* Equation 5.6 and Equation 5.7 are evaluated to determine clock gating variation on the overall consumption, which is -1.04%
* Clock gating is more beneficial than power gating determining, overall, a larger power saving. Thus LR_5 is added to *CG_set*.

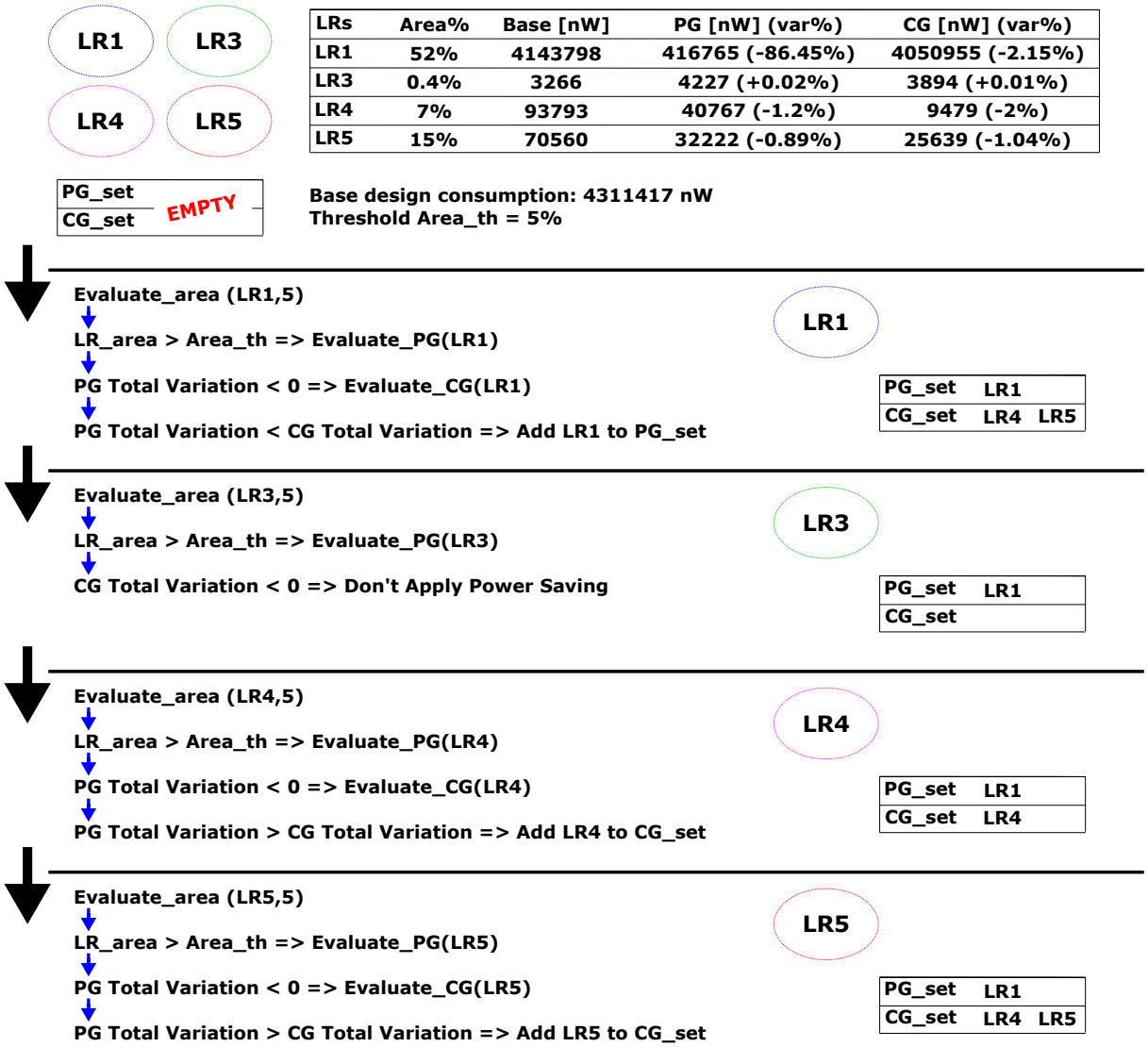


Figure 5.2: Step-by-step example of the enhanced MDC power extension implementing Algorithm 4. Table on the top of the figure reports, for each logic region (*LR*), the power consumption of the region when no powere saving techniques are applied (Base), the power consumption of the *LR* estimated for a perspective power gating application (PG), and the power consumption of the *LR* estimated for a perspective clock gating application (CG). Data in brackets (*var%*) report the percentage variation in the power consumption, with respect to the Base consumption, when PG or CG are applied.

The resulting hardware design with the hybrid application of clock gating and power gating is shown in Figure 5.3. The power gating is applied only to region LR_1 (called PD1 in the figure), while the clock gating is applied to regions LR_4 and LR_5 (called CD4 and CD5 in the figure); the SBoxes *SB_0* and *SB_1* included in region LR_4 are purely combinatorial, so they are not affected by the application of the clock gating. All the remaining logic, which includes region LR_3, is always on.

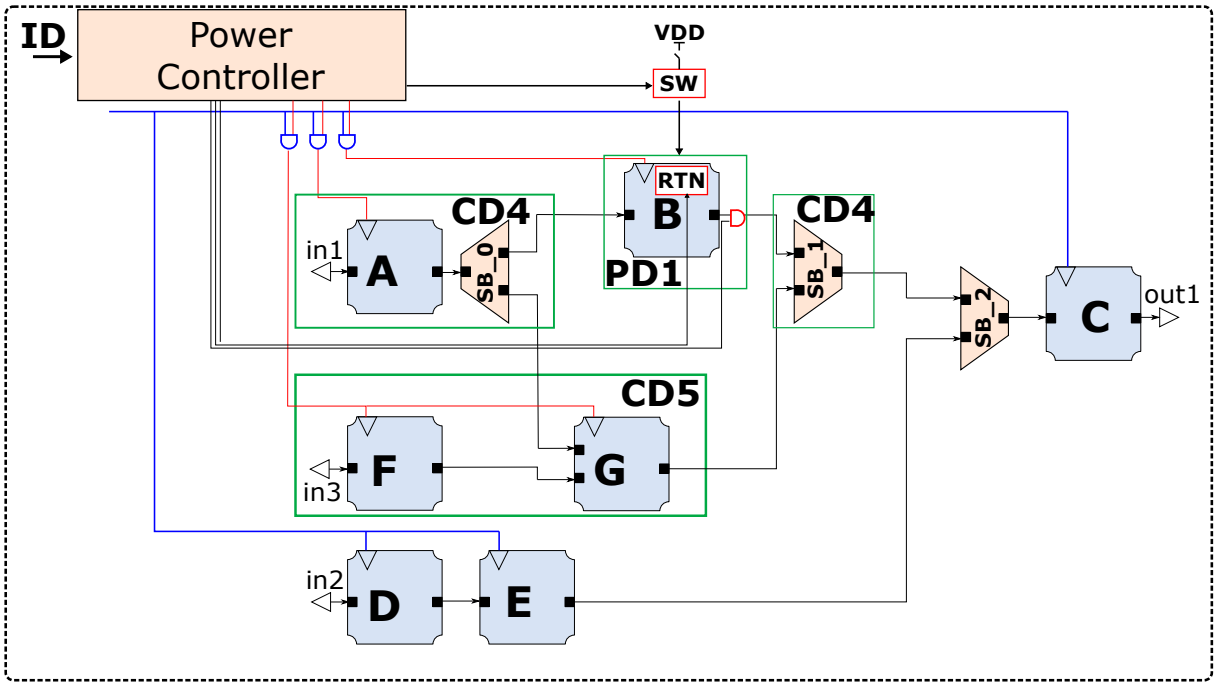


Figure 5.3: Enhanced MDC design suite: Hardware platform with hybrid application of clock gating and power gating methodologies.

5.4 Assessment

This Section discusses two different use cases (in terms of behaviour and resulting power consumption contributions) adopted to assess the presented power estimation flow and the effectiveness of the hybrid clock and power gating management. The first one deals with a simple FFT algorithm implemented on a 90 nm CMOS technology and it has been mainly adopted to evaluate in details the proposed flow. The second one presents a more complex scenario. An image co-processing unit, accelerating a zoom application, has been implemented both on a 90 nm and on a 45 nm CMOS technology in order to assess the robustness of the proposed flow with different technology parameters.

5.4.1 Evaluation Phase - Fast Fourier Transform Algorithm

This section deeply discusses the results obtained considering the Fast Fourier Transform (FFT) use case targeting a 90 nm CMOS technology. FFT is an optimised algorithm for the Discrete Fourier Transform (DFT) calculation (see Equation 5.8). It is widely adopted in several applications, ranging from the solving of differential equations to the digital signal processing.

$$X_k = \sum_{n=0}^{N} x_n e^{-i2\pi k \frac{n}{N}}, k = 0, 1, ..., N-1 \tag{5.8}$$

The FFT algorithm adopted for this use case has been proposed by Cooley and Turkey [26]. It aims at speeding up the calculation of a given size N DFT by considering smaller DFTs of size r, called radix. To obtain the whole original DFT, M stages of size r DFTs are required, where $N = r^M$. Small DFTs have to be multiplied by the so called twiddle factors, according to the decimation in time variant of the algorithm. The simplest DFT with two input and two outputs (radix $r = 2$) takes the name of *butterfly*, by their block scheme. The equations describing a butterfly are:

$$\begin{aligned} X_0 &= x_0 + x_1 w_n^k \\ X_1 &= x_0 - x_1 w_n^k \end{aligned} \tag{5.9}$$

where X_0 and X_1 are the outputs, while x_0 and x_1 are the corresponding inputs. w_n^k are the twiddle factors, defined as:

$$w_n^k = e^{-i2\pi k \frac{k}{N}} \tag{5.10}$$

where k and n are integers depending on the butterfly position in the FFT.

The adopted use case involve a radix-2 FFT of size 8, as depicted in Figure 5.4, obtained by means of three stages involving four butterflies each, meaning 12 butterflies overall ($r = 2$, $M = 3$, $N = r^M = 2^3 = 8$). Stages have been pipelined to keep the system critical path short. The baseline 12 butterflies design then requires three clock periods for the outputs elaboration.

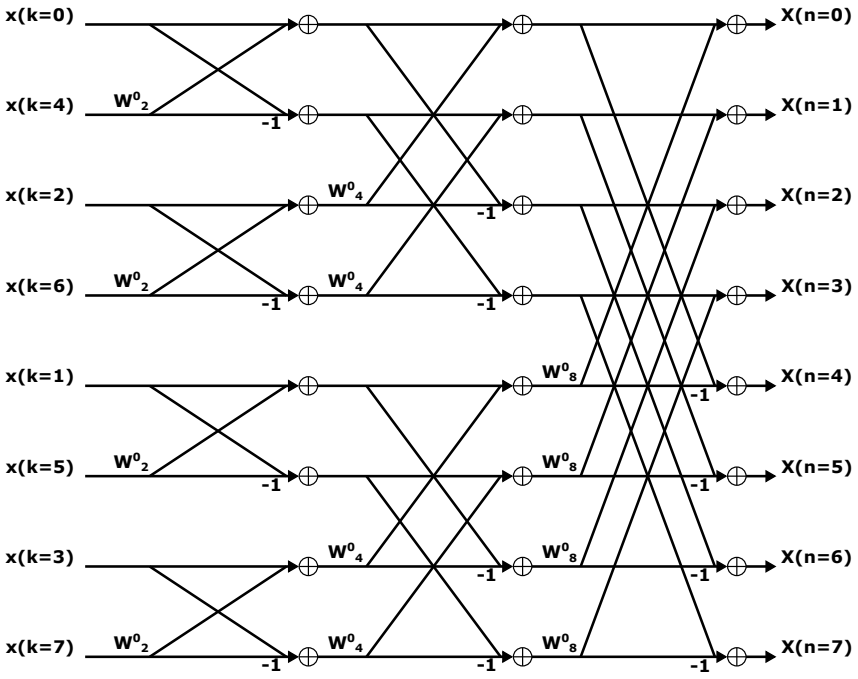


Figure 5.4: FFT use case: Original design with 12 radix-2 butterflies for an FFT of size 8. Twiddle factors w_n^k are calculated according to Eq. 5.10

From the baseline 12 butterfly design, three additional variants have been derived through the decrease of the involved butterflies number. In such a way, the available resources of the design must be multiplexed in time and reused. Therefore, the overall computation latency increases and the throughput becomes lower. The considered 8 FFT configurations are:

- *12b* is the baseline 12 butterflies FFT design, taking 3 clock periods to finalize the transform;
- *4b* involves 4 butterflies for an overall execution latency of 6 cycles;

- *2b* involves 2 butterflies for an overall execution latency of 12 cycles;
- *1b* involves 1 single butterfly for an overall execution latency of 24 cycles.

FFT Coarse-Grain Reconfigurable System Implementation

The above mentioned configurations have been modelled as DPNs graphs [2] and the corresponding CGR system has been assembled with MDC. In this reconfigurable design 8 different *LRs* are identified. Table 5.5 depicts the activation percentage, resource utilization (in terms of activated *LRs* and percentage area) and power consumption of each FFT variant. In general, the higher is the number of butterflies, the more is the corresponding active logic and dissipation.

Table 5.5: FFT use case: Features of the different configurations integrated on the CGR design. Data refer to a 90 nm CMOS target technology.

FFT configuration	T_{ON}	*LRs*	Area%	Static [nW]	Internal [nW]
1b	0.42	(2, 6, 8)	12.86	1750407	1307259
2b	0.21	(2, 5, 7, 8)	20.81	1752106	1539855
4b	0.04	(2, 3, 4, 7)	35.28	1757485	2020953
12b	0.33	(1, 3, 7, 8)	98.03	1776056	2411257

The main purpose of the resulting CGR system is to enable several tradeoff levels between power dissipation and throughput, as illustrated in Figure 5.5. Such a system is capable of dynamically switching among the different configurations, fitting to external environment requests. For instance, in a battery operated environment, when the battery level becomes lower than a given threshold, some throughput can be waived to consume less power.

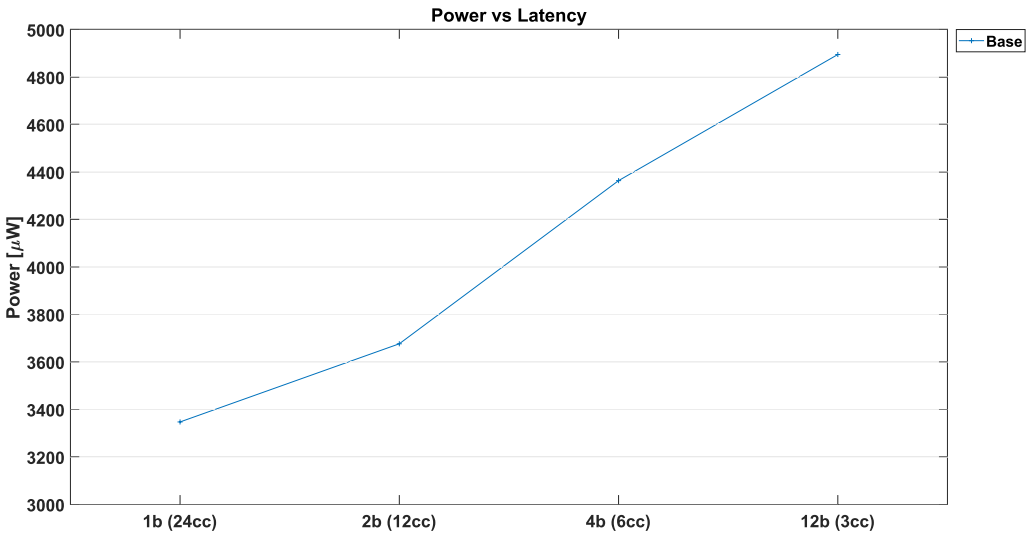


Figure 5.5: FFT use case: Latency versus power consumption tradeoff for the 4 different 8-size FFT configurations.

[2] As explained in Section 3.1 MDC is able to consider a DPNs a generic graph, implementing any kind of communication among actors. In the considered use case, the FFT DPNs are indeed considered as generic graphs, and actors communicate through a handshake communication protocol

Characteristics of each *LRs* are reported in Table 5.6. In this table, given any *LR*, its activation factor (T_{ON}) has been obtained summing up the activation factors of the FFT configurations activating the same region (provided in Table 5.5). For example, LR_2 is activated by *1b*, *2b* and *4b*. Its T_{ON} is 0.67, which is the sum of $T_{ON}(1b) = 0.42$, $T_{ON}(2b) = 0.21$ and $T_{ON}(4b) = 0.04$.

Table 5.6: FFT use case: Logic regions architectural and functional characteristics.

	LR_1	LR_2	LR_3	LR_4	LR_5	LR_6	LR_7	LR_8
T_{ON}	0.33	0.67	0.37	0.04	0.21	0.42	0.58	0.96
$\#iso(e)$	1024	1112	512	2324	1948	1756	256	5259
$\#rtn$	1024	518	256	0	0	0	128	512
$\#reg$	1024	518	256	0	0	0	128	512
$\#iso(r)$	1024	1032	512	2306	1926	1740	256	4934

Power Modelling and Hybrid Power Management Assessment

As explained in Section 5.2, the proposed flow requires a preliminary synthesis of the baseline (without any implemented power saving strategy) CGR system, to retrieve parameters for each *LR*. Figure 5.6 depicts occupancy and logic composition (combinatorial and sequential contributes) of the 8 *LRs*. The area is given in terms of percentage with respect to the overall system area. The biggest (LR_1) occupies more than 60% of the whole system area and it is has the main impact on power consumption. Furthermore, it is quite entirely combinatorial (99.18%), which means that less than 1% of the region is composed by sequential logic, so that power gating should be a very suitable strategy for this *LR*. By Figure 5.6 it is possible to notice that the other *LRs* are much smaller than *LR1*. For all these regions, the proposed power modelling strategy can be extremely beneficial to investigate if power saving techniques may lead or not to an effective power saving. Figure 5.6 also suggests that LR_4, LR_5 and LR_6 cannot benefit from clock gating, since they are fully combinatorial.

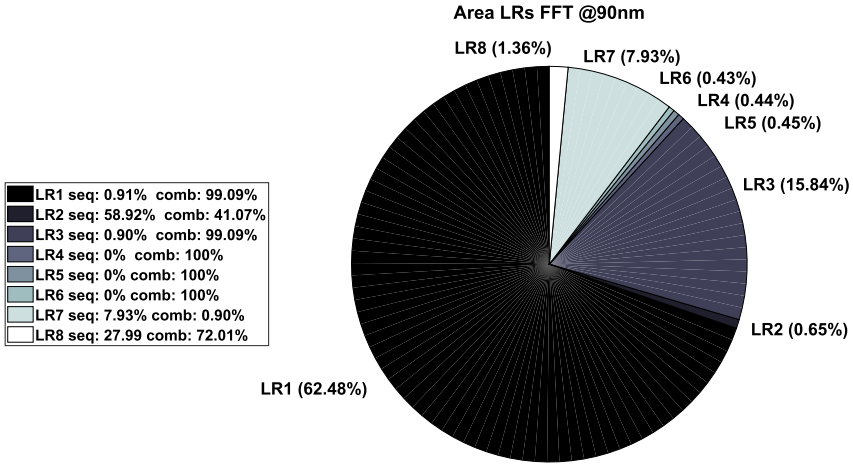


Figure 5.6: FFT use case: Area percentage per *LR*.

Figure 5.7 and Figure 5.8 compare the estimated and measured (retrieved from the post-synthesis reports) power variation, respectively, due to clock gating and power gating. In both cases the reported power refers static and internal contributions, as taken into consideration by the power model.

The remaining term, the net one, is neglected. The error of neglecting the net contribution is discussed in Section 5.4.1. The proposed power models are generally able to accurately approximate the power saving strategy overhead. As expected, LR_1 is the region with the highest power saving, regardless the considered strategy (please note that a negative variation implies a saving in power). It is interesting to notice that LR_8, despite being one of the smallest regions, does not provide any saving if power gated, but it can achieve a little power reduction when clock gated.

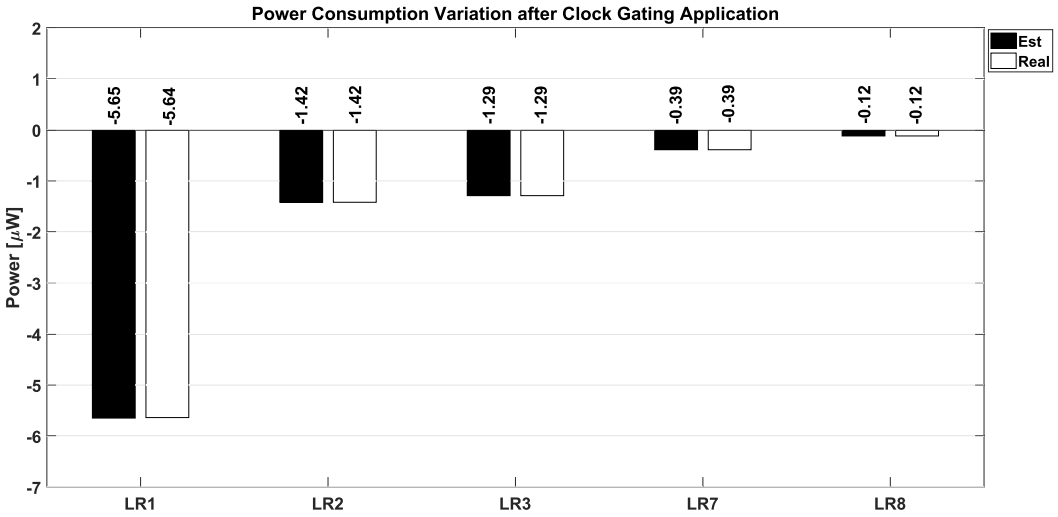


Figure 5.7: FFT use case: Comparison between the estimated and real power variation due to the clock gating integration.

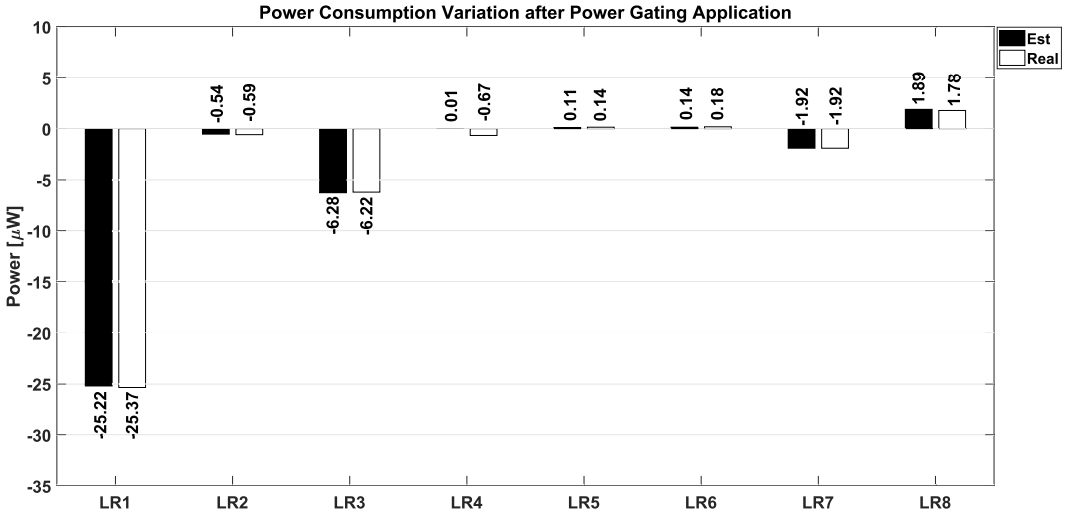


Figure 5.8: FFT use case: Comparison between the estimated and real power variation due to the power gating integration.

The static and internal power variation obtained by applying Equation 5.6 and Equation 5.7, considering a possible clock gating implementation (column *CG Variation %*) and Equation 5.3 and Equation 5.5 for a prospective power gating implementation (column *PG Variation %*), are shown in

Table 5.7. Please notice that clock gating static power variation is not appreciable, since one single clock gating cell is required per *LR*.

Table 5.7: FFT use case: Detailed static and dynamic power variation due to clock gating (*CG Variation %*) and power gating (*PG Variation %*).

LR	*CG Variation %*		*PG Variation %*	
	Static	Internal	Static	Internal
LR_1	0.00	-11.33	-40.87	-9.47
LR_2	0.00	-2.85	+0.23	-1.32
LR_3	0.00	-2.59	-10.44	-2.11
LR_4	—	—	-0.08	0.10
LR_5	—	—	0.08	0.13
LR_6	—	—	0.11	0.17
LR_7	0.00	-0.79	-3.44	-0.39
LR_8	0.00	-0.25	1.42	2.35

Algorithm 4 of the presented methodology implies a preliminary *area thresholding* step. Two different thresholds have been considered for the algorithm evaluation:

- DAT_5%: Threshold set to 5%. Regions with area above the 5% are LR_1, LR_3 and LR_7, so that the *power gating variation estimation* step is performed for each of them. All the considered regions lead to an overall saving (negative variation) larger than those achievable with a prospective clock gating implementation; therefore, they are selected as eligible regions for power gating. *Clock gating variation estimation* is performed on all the remaining sub-threshold regions. The regions capable of providing saving, when clock gated, are LR_2 and LR_8, since LR_4, LR_5 and LR_6 are fully combinatorial. Thus, clock gating will be implemented only on LR_2 and LR_8.
- DAT_10%: Threshold set to 10%. Only LR_1 and LR_3 are above the area threshold and, as occurred also for DAT_5%, they both achieve power saving if implemented with power gating strategies. In this second case, the *clock gating variation estimation* step is performed also on LR_7, which results in a negative variation. Then, the regions to be clock gated are LR_2, LR_7 and LR_8, while LR_4, LR_5 and LR_6 are again discarded.

To assess the proposed flow five designs have been assembled.

- Base: the baseline CGR design without any power saving.
- CG_full: the CGR design, where clock gating is applied blindly to all the regions.
- PG_full: the CGR design, where power gating is applied blindly to all the regions.
- DAT_5%: derived with the proposed automated flow capable of hybrid power and clock gating support, setting the Area Threshold to 5% in Algorithm 4.
- DAT_10%: derived with the proposed automated flow capable of hybrid power and clock gating support, setting the Area Threshold to 10% in Algorithm 4.

These designs have been synthesized with Cadence RTL Compiler, targeting the same 90 nm CMOS technology adopted to synthesise and simulate the baseline CGR design, which results have fed the *Power Analysis* block of the proposed enhanced power management flow to assemble DAT_5% and DAT_10%. Figure 5.9 depicts power consumption (internal, static and total) of these designs. The

power consumption data are reported by Cadence RTL Compiler, considering the synthesised designs and their switching activity from post-synthesis simulations. Since LR_1 occupies more than the 60% of the design area and it is mainly combinatorial, little differences among the entirely power gated design (PG_full) and the hybrid clock and power gated ones (DAT_5% and DAT_10%) are visible. Nevertheless, DAT_5% achieves the largest power saving (-45.12%) among all the designs, validating the proposed hybrid and selective management with respect to a one-fit-to-all solution. CG_full, capable of diminishing only the dynamic power consumption, is the worst design among those applying power management.

The area overhead of the implemented power management strategies, reported in the legend of Figure 5.9, proves that the proposed hybrid management leads to less area hungry designs than the entirely power gated one. In fact, DAT_5% and DAT_10% present half of the area overhead of PG_full. CG_full data confirms that clock gating has a very little impact on the baseline design, presenting a negligible area overhead (two orders of magnitude smaller than DAT_5% and DAT_10%).

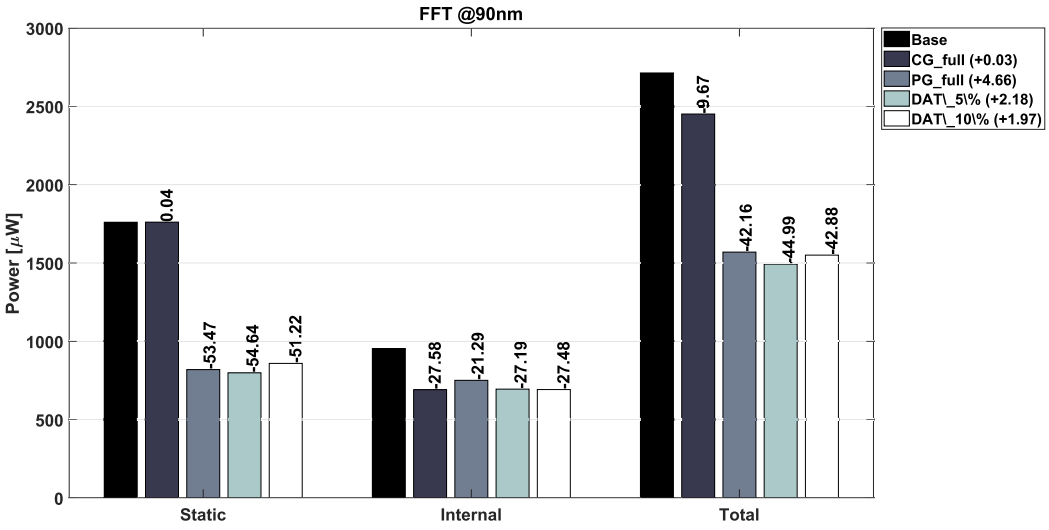


Figure 5.9: FFT use case: Comparison between the base design and the four gated designs. Legend shows, in brackets, the power management area overhead for each design wrt to the Base one.

For the sake of completeness, in Figure 5.10 the tradeoff levels between power and latency (and, in turn, throughput) are illustrated for all the considered designs. The tradeoff curves demonstrate that power management strategies are generally extremely beneficial within a CGR scenario.

Accuracy and Errors

Table 5.8 and Table 5.9, respectively considering the *clock gating variation estimation* step and the *power gating variation estimation* step of Algorithm 4, assess the accuracy of the presented power modelling approach. These tables, in each row, report the estimation errors with respect to the real consumption of the baseline CGR system, where the given power saving strategy (i.e. clock gating in Table 5.8 and power gating in Table 5.9) is applied only to the *LR* specified in the first column. Table 5.8 depicts an overview of the estimations accuracy for the clock gating variation. Estimations are really accurate, being the error on the clock gating variation always below the 0.3%. The error due to the clock gating cells overhead is very limited too, being always under the 1.1%.

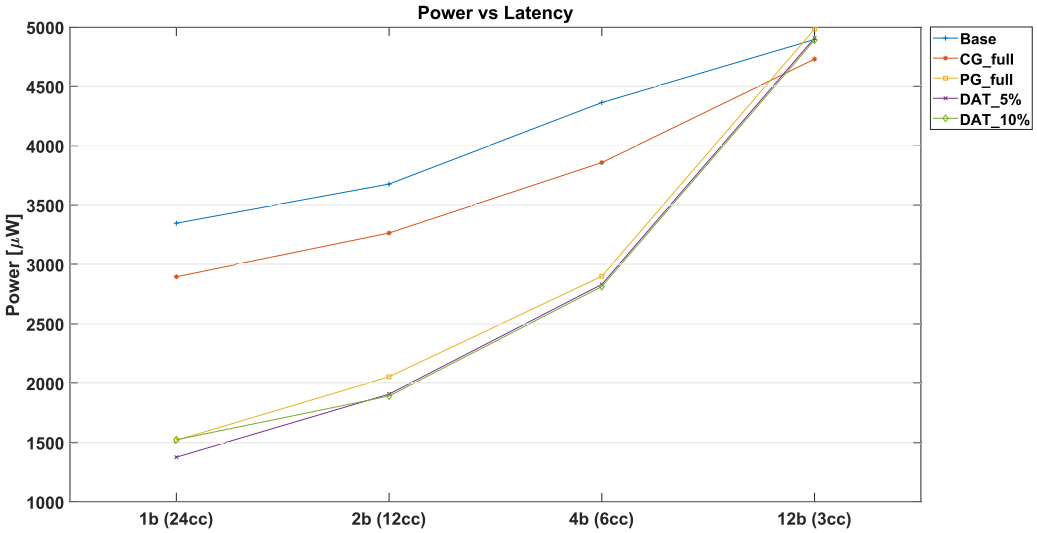


Figure 5.10: FFT use case: Latency versus power consumption tradeoff for the 4 different 8-size FFT configurations, when gated designs are adopted.

Table 5.8: FFT use case: *Clock gating variation estimation* step accuracy.

LR	CG variation %			Overhead	Net
	Est.	Real	Err.%	CG_{cell} Err.%	Err.%
LR_1	-5.73	-5.77	0.06	0.83	0.18
LR_2	-1.42	-1.42	0.12	1.03	0.40
LR_3	-1.34	-1.38	0.05	0.84	0.21
LR_7	-0.39	-0.39	0.05	0.96	0.22
LR_8	-0.12	-0.12	0.29	1.08	0.47

Table 5.9: FFT use case: *Power gating variation estimation* step accuracy.

LR	PG variation %			Overhead Error %			Net
	Est.	Real	Err.%	Iso	Rtn	Contr.	Err.%
LR_1	-25.22	-25.37	0.59	5.18	4.64	0.75	10.3
LR_3	-6.28	-6.22	1.07	16.34	2.97	0.76	12.6
LR_7	-1.92	-1.92	0.24	9.24	1.08	0.83	13.6

Also power estimations per *LR*, reported in Table 5.9, demonstrate to be very accurate, leading to errors that are always below 1.1%. Also errors related to the estimation of state retention and power controller overhead are quite low (respectively below 5% and 1%). The isolation cells overhead estimation is less precise, resulting in an error of 16.36% for PD3, due to the fact that the static and internal values of $P(ISO_{ON})$ and $P(ISO_{OFF})$ are characterized as average values, the same for each *LR*. Nevertheless, this error has no visible impact on the total estimation one that is 1.07%.

Table 5.8 and Table 5.9 report also the errors caused by omitting the power net term (column *Net %(Err.)*) in Equation 5.7 and Equation 5.5. This error is obtained by comparing the estimated variation (not comprehensive of the net contribute) with the real measured variation comprehensive of the net term, as extracted by the synthesis reports. The net error is higher in the power gating variation estimation (13.6% for LR_7) with respect to the clock gating one (at maximum 0.47% for LR_8),

Table 5.10: Zoom Co-processor use case: Computational kernels distinctive features.

kernel	# actors	T_{ON}	functionality	*LRs*
abs	1	0.03	absolute value calculation	(4)
chgb	7	0.33	bilevel/grayscale block checking	(5 6 7 12 13)
cubic	10	0.06	linear combination calculation	(1 5 9 10 13)
cubic_conv	6	0.09	cubic filter convolution	(5 7 8 9)
median	9	0.06	median calculation	(1 3 5 11 13)
min_max	1	0.01	maximum/minimum finding	(11)
sbwlabel	17	0.42	edge block checking	(2 4 5 6 13)

since power gating requires to add in the design several extra logic than a clock gating, less invasive, implementation.

5.4.2 Validation Phase - Zoom Application

This Section presents validation of the presented approach on a second use case, targeting the same 90 nm technology used for the FFT use case and a smaller 45 nm library. The reconfigurable computing core of an image co-processing unit, accelerating a zoom application, has been assembled. The zoom application is meant to scale an image according to the given zooming factor. Missing pixels of the zoomed image are calculated by adaptively interpolating the neighbouring values. Seven computational kernels have been identified and modelled as DPNs. These kernels have been combined by MDC to obtain a multi-dataflow specification, constituting the computing core of the CGR accelerator in charge of accelerating the zoom application. Thirteen *LRs* are identified on the CGR zoom co-processor. Table 5.10 summarizes the kernels composition (in terms of number of dataflow actors and activated *LRs*), activation factor and main functionality. The main difference between this scenario and the FFT one is that in the zoom co-processor it is not necessary to retain the status of any kernel when switching among them. This means that, applying power gating, no retention cells are needed in the identified regions.

Zoom Co-processor Validation Results at 90 nm CMOS Technology

This section provides the discussion of the achieved results in the zoom co-processor scenario using the same 90 nm CMOS technology adopted for the FFT CGR designs assessment. From the implementation point of view the same designs considered for the FFT use case are discussed:

- Base: the baseline CGR design without any power saving.
- CG_full: the CGR design, where all the 13 *LRs* are clock gated.
- PG_full: the CGR design, where all the 13 *LRs* are power gated.
- DAT_5%: the CGR design, where hybrid power and clock gating support is implemented by means of the proposed flow, setting the Area Threshold to 5% in Algorithm 4. Composition of DAT_5% is depicted in Table 5.11.
- DAT_10%: the CGR design, where hybrid power and clock gating support is implemented by means of the proposed flow, setting the Area Threshold to 10% in Algorithm 4. Composition of DAT_10% is depicted in Table 5.11.

Table 5.11: Zoom Co-Processor at 90 nm CMOS technology: Characterization of the hybrid, clock and power gated designs, achieved with the proposed automated flow.DAT_5%: area threshold 5%. DAT_10%: area threshold 10%. NA stands for not assigned and includes those *LRs* that placed in the always-ON domain.

design	>Th	PG_set	CG_set	NA
DAT_5%	LR_1 LR_2 LR_3 LR_6 LR_8 LR_{10} LR_{12} LR_{13}	LR_1 LR_2 LR_3 LR_8 LR_{10} LR_{12}	LR_4 LR_6 LR_7 LR_9 LR_{11} LR_{13}	LR_5
DAT_10%	LR_1 LR_8	LR_1 LR_8	LR_2 LR_3 LR_4 LR_6 LR_7 LR_9 LR_{10} LR_{11} LR_{12} LR_{13}	LR_5

Figure 5.11 depicts static, internal and total power consumption for each considered design. In this case, the differences among CG_full, PG_full, DAT_5% and DAT_10% are not so evident. The reason is that, in this scenario, the dynamic power consumption (due to the internal power) is considerably higher than the static one. As visible in the reported histograms, on average there are approximately more than two orders of magnitude of difference between the two contributions.

Clock gating and power gating demonstrate to be equally capable of cutting down the internal power consumption. LR_5 is the only region that Algorithm 4 completely discards by any form of power management, both in the DAT_5% design and in the DAT_10% one. It is fully combinatorial; therefore, clock gating does not provide any positive effect on it. Nevertheless, it is so small (0.65% of the whole system area) that, if power gated, it cannot provide any substantial benefit. A closer observation of the histograms confirms what we already got for the FFT: despite the similar trend for all the designs, which lead to more than the 62% of power saving, DAT_5% consumes less than any other (62.61% of saving), while the CG_full design is the less beneficial (62.29% of saving).

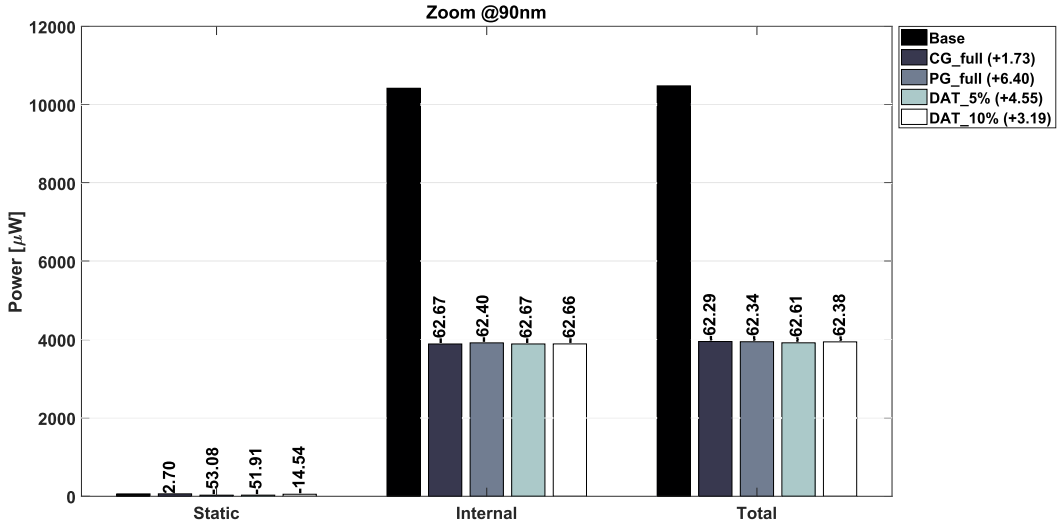


Figure 5.11: Zoom Co-Processor at 90 nm CMOS technology: Comparison between the base design and the four gated designs. Legend shows, in brackets, the power management area overhead for each design wrt to the Base one.

Focusing on the static histograms, the CG_full design introduces a small overhead with respect to Base. That is due to the 12 (one for each region but LR_5) clock gating cells introduced in the always-ON domain of this design, which never contribute to save any static power consumption. When power gating is applied there is always a benefit in terms of static power consumption: DAT_5% saving is slightly higher than the PG_full one, being both over 51%; DAT_10% is still beneficial, but its saving is limited to the 15%. Please note that the difference between DAT_5% and DAT_10% (in terms of static consumption) demonstrates that, in the *Area Threshold* step of the proposed Algorithm 4, it is better to opt for small area threshold values to achieve higher saving results. In terms of area occupancy, reported in the legend of Figure 5.11, the PG_full design is the one with the largest overhead, +6.4%. DAT_5%, which is the most beneficial in terms of power, shows a slightly smaller overall overhead, +4.55% of the whole system area. The CG_full is again the less invasive one, leading just to +1.73% of area overhead.

Summarizing, DAT_5% constitutes the optimal solution for the Zoom co-processor scenario, considering a 90 nm technology. DAT_10%, which is less beneficial than DAT_5% in saving static power consumption, is a better solution than a fully power gated design, presenting basically the same power saving (-62.38% for DAT_10% vs. -62.29% for PG_full) but a smaller area overhead (+3.19% for DAT_10% vs. +6.4% for PG_full).

Table 5.12: Zoom Co-Processor at 90 nm CMOS technology: clock gating variation estimation step and *power gating variation estimation* step accuracy.

LR	CG saving %			Net	PG saving %			Net
	Est.	Real	Err.%	Err.%	Est.	Real	Err.%	Err.%
LR_1	-6.307	-6.313	0.09	0.17	-6.322	-6.331	0.15	0.18
LR_2	-23.369	-23.373	0.02	1.67	-23.464	-23.463	0.00	1.26
LR_3	-6.507	-6.514	0.10	1.59	-6.537	-6.544	0.10	1.65
LR_4	-0.958	-0.964	0.68	1.34	—	—	—	—
LR_5	—	—	—	—	—	—	—	—
LR_6	-0.870	-0.878	0.81	1.22	—	—	—	—
LR_7	-1.033	-1.039	0.63	0.15	—	—	—	—
LR_8	-6.987	-6.986	0.01	1.83	-7.062	-7.058	0.05	2.02
LR_9	-2.436	-2.443	0.27	1.55	—	—	—	—
LR_{10}	-5.474	-5.480	0.11	1.57	-5.498	-5.503	0.10	1.65
LR_{11}	-3.329	-3.285	1.32	3.67	—	—	—	—
LR_{12}	-4.267	-4.273	0.15	1.86	-4.278	-4.288	0.23	1.83
LR_{13}	-0.649	-0.656	1.01	1.13	—	—	—	—

Table 5.12 reports the estimation error of the proposed automated hybrid power management design flow, when the power saving percentages for the considered domains, respectively considering clock gating (*clock gating variation estimation*) and power gating (*power gating variation estimation*), are evaluated. *CG saving %* errors are do not exceed 1.5% and *PG saving %* errors are always below 0.3%. These data confirm the accuracy of the proposed models, as in the FFT use case. Table 5.12, for both clock and power gating implementations, depicts also the error of neglecting the net term in the dynamic power consumption. Again, as in the previously discussed scenario, models are not affected by this simplification.

Zoom Co-processor Validation Results at 45 nm CMOS Technology

In order to provide a robust validation of the proposed approach, this Section present assessment on the the same zoom co-processor designs, targeting a 45 nm CMOS technology.

The implemented designs are the same as in Section 5.4.2. However, targeting a smaller technology and having already established that the 10% area threshold leads to power results comparable to those of the fully power gated design, in this second trial an additional design is considered: DAT_1%. Setting the area threshold to 1% quite all the *LRs* are considered for a prospective power gating implementation (Please refer to Table 5.13 for the composition of DAT_1%, DAT_5% and DAT_10%). Here follows the list of the implemented designs.

- Base: The same as in the 90 nm synthesis trial.
- CG_full: The same as in the 90 nm synthesis trial.
- PG_full: The same as in the 90 nm synthesis trial.
- DAT_1%: the CGR design, where hybrid power and clock gating support is implemented by means of the proposed flow, setting the Area Threshold to 1% in Algorithm 4.
- DAT_5%: The same as in the 90 nm synthesis trial.
- DAT_10%: The same as in the 90 nm synthesis trial.

Table 5.13: Zoom Co-Processor at 45 nm CMOS technology: Characterization of the hybrid, clock and power gated designs, achieved with the proposed automated flow.DAT_5%: DAT_1%: area threshold 1%. area threshold 5%. DAT_10%: area threshold 10%. NA stands for not assigned and includes those *LRs* that placed in the always-ON domain.

design	>Th	PG_set	CG_set	NA
DAT_1%	LR_1 LR_2 LR_3 LR_4 LR_6 LR_7 LR_8 LR_9 LR_{10} LR_{11} LR_{12} LR_{13}	LR_1 LR_2 LR_3 LR_4 LR_7 LR_8 LR_9 LR_{10} LR_{11} LR_{12}	LR_6 LR_{13}	LR_5
DAT_5%	LR_1 LR_2 LR_3 LR_6 LR_8 LR_{10} LR_{12} LR_{13}	LR_1 LR_2 LR_3 LR_8 LR_{10} LR_{12}	LR_4 LR_6 LR_7 LR_9 LR_{11} LR_{13}	LR_5
DAT_10%	LR_1 LR_8	LR_1 LR_8	LR_2 LR_3 LR_4 LR_6 LR_7 LR_9 LR_{10} LR_{11} LR_{12} LR_{13}	LR_5

Figure 5.12 illustrates power consumption in terms of static, internal and total contributions. The dynamic power consumption is still higher than the static one, determining the overall trend of the total power. However, with the scaling of the channel length, the ratio among internal and static power on average has decreased from a factor of 100 to approximately 10. In this second trial, the influence of the static power consumption is partially reflected on the total one. Technology scaling and the different static versus dynamic power ratio are such that PG_full is capable of providing better overall saving results than DAT_5% and DAT_10%. At 45 nm technology, designers are required to select a very low area threshold in Algorithm 4 to achieve really optimal results. DAT_1%, which basically excludes from power gating only the 3 *LRs*, saves up to 61.84% of total Base power and represents the

optimal design solution for the zoom co-processor in this second synthesis trial. Please note also that, lowering down the area threshold, the area of the optimal design and that of the fully power gated one are pretty similar. It is possible to conclude that as technology gets smaller the *Area Threshold* step of the proposed algorithm is less beneficial still, in the automated flow, its presence makes the overall process more robust, avoiding useless iterations on not convenient by construction designs when the technology are not so constrained or the ratio among static and dynamic consumption is larger.

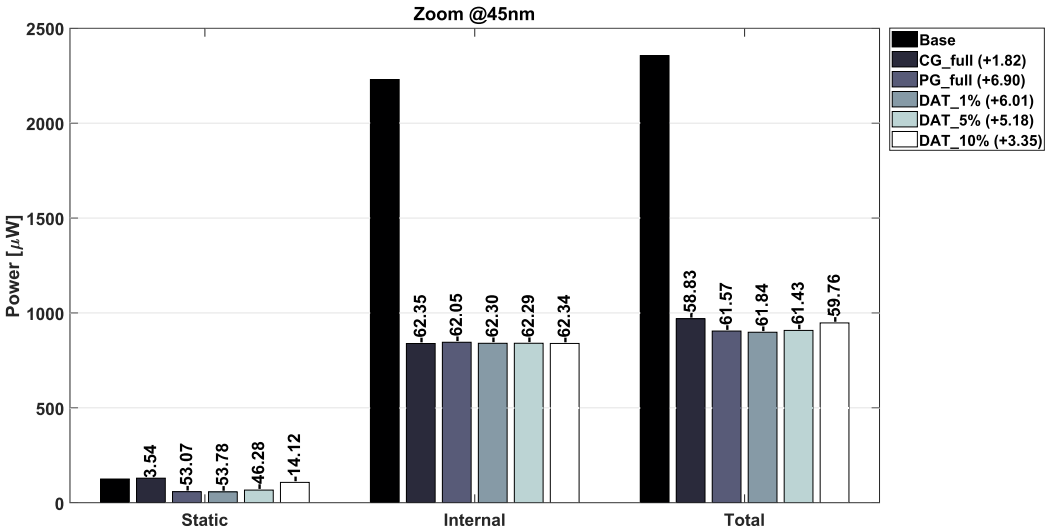


Figure 5.12: Zoom Co-Processor at 45 nm CMOS technology: Comparison between the base design and the five gated designs. Legend shows, in brackets, the power management area overhead for each design wrt to the Base one.

The accuracy of the proposed models, targeting the 45 nm CMOS technology, is reported in Table 5.14, which contain both clock gating and power gating estimation errors. The models, even neglecting the net contribution in the discussed equations, are extremely accurate (the error never exceeds 3.70%).

5.4.3 Power switch overhead

The sleep transistors are inserted in the design during the place and route process and their overhead is strictly use-case dependent since it is related to the aspect ratio of the macro and to the style of power routing that is selected in the target design. Since the proposed power estimation model is based on the synthesis of the design, the contribution of these cells is not considered yet.

The insertion of header/footer switches determines two kinds of power overhead: (1) a static power-related overhead; (2) the dynamic power dissipated during sleep and wake-up transition. Another overhead that have to be taken into consideration is the time necessary to wake-up the power domain. For the proper operation of the power gating methodology, the gating logic has to be enabled/disabled according to a switch on/off protocol [90] that requires a well defined transition sequence (see Section 4.3). This overhead can be neglected only if the computation time of the application is much higher (at least one order of magnitude) than the switch on/off sequence. However, an estimation of the length of the power switches chain is necessary to precise determine the wake-up time.

The static power-related contribution is fixed by the technology library and it is always present regardless the ON/OFF state of the power domain. It could be inserted in Equation 5.3 as: $P_{lkg}(SW) *$

Table 5.14: Zoom Co-Processor at 45 nm CMOS technology: *Power gating variation estimation* step and *clock gating variation estimation* step accuracy.

LR	CG saving %			Net	PG saving %			Net
	Est.	Real	Err.%	Err.%	Est.	Real	Err.%	Err.%
LR_1	-5.507	-5.501	0.13	0.53	-5.686	-5.699	0.23	0.47
LR_2	-22.063	-22.029	0.15	0.81	-22.990	-22.982	0.03	1.02
LR_3	-6.273	-6.262	0.18	0.91	-6.569	-6.566	0.04	1.13
LR_4	-0.919	-0.917	0.22	1.52	-0.946	-0.944	0.15	1.82
LR_5	—	—	—	—	—	—	—	—
LR_6	-0.853	-0.833	0.26	1.59	-0.747	-0.748	0.25	0.25
LR_7	-0.935	-0.933	0.21	1.49	-0.955	-0.957	0.17	1.36
LR_8	-6.724	-6.714	0.16	0.66	-7.464	-7.460	0.06	1.36
LR_9	-2.350	-2.345	0.18	1.09	-2.475	-2.482	0.25	1.10
LR_{10}	-5.270	-5.261	0.17	0.91	-5.473	-5.470	0.04	1.09
LR_{11}	-3.257	-3.205	1.63	2.04	-3.422	-3.361	1.79	2.58
LR_{12}	-4.176	-4.169	0.17	0.98	-4.327	-4.330	0.08	1.19
LR_{13}	-0.623	-0.621	0.28	1.85	-0.587	-0.580	1.25	3.70

$\#switches$ where $P_{lkg}(SW)$ is the static power consumption of the considered power switch, as reported in the technology library, and $\#switches$ is the number of power switches inserted in the power domain. The dynamic power contribution is only relevant when intervals between successive kernel switches are in the order of tens of cycles (Hu et al. [47]). When the computation of the kernels last tens of cycles also the wake-up time is not relevant. Thus, in designs with low switching rates, these two overhead contributions could be neglected.

The FFT use-case is a really simple design, used only for the development of the power estimation model and, as reported in Section 5.4.1, its kernels are far from lasting tens of cycles. The Zoom application adopted for the validation phase of the proposed model is a real use-case but it is a small size design, where the execution of the fastest kernels lasts 24 clock cycles. Considering a bigger and more complex real use-case, such as interpolation filtering for motion compensation in High Efficiency Video Coding [34], it is possible to achieve the condition for neglecting the dynamic power consumption of the sleep transistors and the wake-up time overhead. This application involves 2-dimensional filters working on sub-blocks of pictures belonging to the same video sequence. The smallest block, corresponding to the fastest execution time, has 8 x 8 pixels.

5.4.4 Advantages of the proposed approach

Considering a CGR system implementing N different functionalities and partitioned into k different *LRs*, the proposed selection algorithm, based on the power models embodied in Equation 5.3, Equation 5.5, Equation 5.6 and Equation 5.7, requires the synthesis of the baseline CGR design (without any power saving strategy applied) and N hardware simulations, each one running a different functionality (i.e. executing the different DPNs provided as input to the MDC tool). The hardware simulations are needed since the real switching activity is essential for correct dynamic power estimation. Table 5.15, targeting the FFT scenario and the power gating implementation, depicts the estimated and real power overhead when estimations are performed adopting the default synthesis reports (without taking into account the real switching activity). The estimation errors are extremely high when the switching activity is neglected; therefore, the proposed models are not capable of properly determining which *LRs* would actually benefit of power gating.

Table 5.15: FFT use case at 90 nm CMOS technology: *Power gating variation estimation* step accuracy, using reports generated without the real switching activity.

LR	PG variation %		
	Est.	Real	Err.%
LR_1	-30.54	-26.10	17.07
LR_2	-0.19	-1,99	90.22
LR_3	-21.83	-6.95	214.30
LR_4	-0.12	-0.67	80.97
LR_5	-0.06	-0.59	90.11
LR_6	-0.07	0.56	86.60
LR_7	-15.39	-2.64	482.93
LR_8	-0.38	-0.02	1941.81

In order to understand the advantages of the proposed approach, let's compute the effort needed to determine the optimal saving strategy for each region if our flow is not adopted. It is required to:

1. synthesize the baseline design without any power management support;
2. synthesize one power gated design and one clock gated design for each *LR*;
3. perform N different hardware simulations for the baseline design, to retrieve the real switching activity of the system;
4. perform N different hardware simulations for each power gated and clock gated design, to retrieve their real switching activity;
5. compare each power gated design and clock gated design, in the different operating conditions, with respect to the synthesized baseline CGR design.

The presented flow requires only point 1 and point 3. In numbers it corresponds to one single synthesis and N hardware simulations. On the contrary, not using our approach, $2*k+1$ synthesis (k for power gating evaluation, k for clock gating evaluation plus the baseline one) and $N*(2*k+1)$ hardware simulations are necessary. The only simplification that may be done, even without adopting the proposed approach, is when a given region is fully combinatorial. This would save the effort related to its perspective clock gating evaluation.

Dealing with the presented use cases, for the FFT (Section 5.4.1) there are $N = 4$ different functionalities and $k = 8$ *LRs*. Among these latter 3 are fully combinatorial. The proposed approach required one synthesis and 4 hardware simulations, rather than 14 synthesis (8 power gated *LRs*, 5 clock gated *LRs* and the baseline design) synthesis and 56 hardware simulations (4 for each synthesized design). Considering the zoom co-processor (Section 5.4.2), $N = 7$ and $k = 13$, with only 1 fully combinatorial *LR*. The proposed approach required one synthesis and 7 hardware simulation, rather than 26 synthesis (13 power gated, 12 clock gated and the baseline designs) and 182 hardware simulations.

5.5 Chapter Remarks

This Chapter presented an automated methodology capable of estimating, prior any physical implementation, the effectiveness and costs that power gating or clock gating would have when implemented on top of the functional logic regions constituting a CGR system. This methodology is based on static and dynamic power estimation models that, in a separate manner for each logic region in the

design, are capable of determining the power consumption variation due to clock gating and power gating on the basis of the functional, technological and architectural parameters of the baseline system. These models and the corresponding estimation algorithm are applicable in any CGR scenario and are integrated in the MDC tool, improving its basic functionality.

By considering two different scenarios and adopting different ASIC technologies, the assessments proved that the enhanced MDC flow is capable of guiding the designers towards the definition of the optimal power management support. It is more efficient than the previous, blindly applied, methodology and the proposed models turned out to be extremely accurate. Finally, as demonstrated in Section 5.4.4, the new flow drastically reduces the number of designs to be synthesized and simulated, leading to save both designer effort and computational time. However, given that the ratio between the static and the dynamic power consumption in digital systems drastically changes according to the considered technology (the smaller the technology is, the bigger the impact of the static power on total power is), the power gating strategy and, in turn, the presented methodology and power analysis flow is more beneficial for smaller technologies.

List of Publications Related to the Chapter

Journal papers

- Francesca Palumbo, Tiziana FANNI, Carlo Sau, Paolo Meloni, and Luigi Raffo, *Modelling and Automated Implementation of Optimal Power Saving Strategies in Coarse-Grained Reconfigurable Architectures*. Journal of Electrical and Computer Engineering, vol. 2016, Article ID 4237350, 27 pages, 2016. https://doi.org/10.1155/2016/4237350.

Conference papers

- Tiziana FANNI, Carlo Sau, Paolo Meloni, Luigi Raffo and Francesca Palumbo, *Power modelling for saving strategies in coarse grained reconfigurable systems*. 2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig), Mexico City, 2015, pp. 1-4. doi: 10.1109/ReConFig.2015.7393337

- Tiziana FANNI, Carlo Sau, Paolo Meloni, Luigi Raffo, and Francesca Palumbo. 2016. Power and clock gating modelling in coarse grained reconfigurable systems. In Proceedings of the ACM International Conference on Computing Frontiers (CF '16). ACM, New York, NY, USA, 384-391. DOI: https://doi.org/10.1145/2903150.2911713

- Tiziana FANNI and Luigi Raffo, *Coarse grain reconfiguration: Power estimation and management flow for hybrid gated systems*. 2016 International Conference on ReConFigurable Computing and FPGAs (ReConFig), Cancun, 2016, pp. 1-4. doi: 10.1109/ReConFig.2016.7857160

- Tiziana FANNI, "Optimal Implementation of Power Saving Techniques in CGR Systems", Cyber-Physical Systems PhD & Postdoc Workshop 2018. Alghero (Italia). CEUR-WS.org/Vol-2208

Other scientific papers

- Francesca Palumbo, Tiziana FANNI, Carlo Sau, Paolo Meloni and Luigi Raffo, *Automated Flow for Hybrid Clock and Power Gating in Coarse-Grained Reconfigurable Architectures*, at the 2017 Riunione Annuale dell'Associazione SocietÃ Italiana di Elettronica (ex GE), Como (Italy) June 2017.

Chapter 6

Coarse-Grain Reconfiguration on FPGA - Desynchronizing Actors and Clock Gating for Energy Optimization

The modularity and parallelism of dataflow model of computation make them suitable for key aspects of design exploration and optimization, such as efficient scheduling, task synchronization, memory and power management. Several tools for hardware and software design, that exploit dataflow models of computation exist. However, automatic tools are not always the best solution to explore different design optimisation and power management techniques when ad-hoc methods are required. The lightweight dataflow (LWDF) programming methodology provides an abstract programming model that supports dataflow-based design of signal processing components and systems. The research presented in this Chapter integrates the LWDF methodology with hardware description languages (HDLs), and in particular the main contribution of this thesis to LWDF methodology is the application of the HDL-integrated power management techniques to develop efficient methods for low power hardware implementation.

This dataflow-driven methodology helps the designers to rapidly incorporate and efficiently experiment with new optimization techniques for dataflow-based implementations. For these reasons, even if it has not been applied to coarse-grain reconfigurable (CGR) systems yet, LWDF is a promising methodology for the investigation of optimization techniques applied to these systems too. The work presented in this Chapter has been conducted into the context of a collaboration among the Microelectronics and Bioengineering Lab (EOLAB) (University of Cagliari), the Intelligent system DEsign and Applications (IDEA) Lab (University of Sassari),the Department of Electrical and Computer Engineering (University of Maryland at College Park) and the Department of Pervasive Computing (Tampere University of Technology).

6.1 SOA on Power Management in Dataflow-based designs

As explained in Chapter 2, dataflow model of computation provides valuable model-based design properties for signal processing systems, and has been adopted in a wide variety of tools for both soft-

ware and hardware design. Summarizing the ones more related to this thesis, the CAL programming language [36] has been proved to increase the productivity compared to reference code written in traditional (hardware and software) languages, and to be a convenient and portable parallel programming model, suitable for automated tools also when targeting hardware. Indeed the ORCC toolset is a complete development environment, based on the RVC-CAL language accelerates the development of multimedia applications in both software and hardware [130]. As deeply described in Chapter 3, MDC tool is a framework for the automatic creation of CGR platforms, that performs a complete design space exploration, evaluating the trade-off among resource usage, power consumption and operating frequency [82]. While CAPH is a dataflow language and toolchain for the specification of stream-processing applications [98].

Recently different works have explored the deployment of power management techniques, in conjunction with dataflow-based designs. Danelutto et al. [28] propose a methodology for the implementation of power aware dataflow runtime systems. By constantly monitoring the application and by changing the amount of used resources according to the workload condition, they limit the power consumption. Holmbacka et al. [46] present an approach to integrate fast parallel software directly with the power management by injecting performance and parallelism into the software as meta-data to the power manager. The meta-data is extracted by a dataflow programming frame work, PREESM [85], and injected into code segments as a parameter in a Non-Linear Programming optimization problem for minimizing total power. An interface between the applications and the hardware resources is provided in combination with a novel power management runtime system called Bricktop. Madroñal et al. [68] presented the Papify-PREESM code generator, integrating PREESM [85] with the Performance API (PAPI) [115] within a Y-chart design flow, for the generation of automatic instrumented code that integrates performance monitoring counters based on PAPI library into PREESM code generation, to enable transparent actor timing and hardware resource usage profiling for developers.

However only few works focussed on power management strategies for digital hardware implementations. Brunet et al. [13] propose a design and implementation methodology based on the dataflow model of computation for GALS-based applications. The methodology maps the application into multiple clock domains subsequently assigning a clock frequency to each clock domain in order to reduce the overall power consumption while meeting the design performance requirements. Bezati et al. [9] presented an extension of Xronos, a High-Level Synthesis tool, to achieve power savings by selectively switching the clock signal off to the circuits when they are temporarily inactive. The MDC tool has the capability of partitioning the design into a minimum set of logic regions, composed of actors always active together, and applying to them clock gating strategy for both FPGA and ASIC [82].

Generally speaking, these methodologies and tools are limited by the language used to describe the adopted dataflow description or by the generated HDL, which can be target dependent (such as a Xilinx FPGA) but not usable for an application specific integrated circuit (ASIC) flow (e.g., see [13, 9]). Furthermore, these tools support the user-friendly application of existing design optimization techniques, rather than the rapid prototyping of new techniques. Automatic methods and tools require significant effort in development and maintenance of graph analysis and code generation functionality, and may be too costly for models and design approaches that are not mature. System designers must therefore resort to ad-hoc methods to explore design alternatives that span multiple implementation scales, platform types, or dataflow modelling techniques.

The lightweight dataflow (LWDF) programming [103] helps to address this gap by providing a compact set of APIs that can be used to incorporate advanced dataflow techniques in a manner that does not require development and maintenance of automation tools. Rather than being focused on automation, LWDF is designed to help the designer architect an efficient dataflow-based implementation and iteratively experiment with it. This capability allows the designer to rapidly incorporate and experiment with advanced power optimization techniques in the framework of a systematic

dataflow-based design methodology. At the same time, because the LWDF APIs are based on formal dataflow principles, LWDF-based implementations can be well-suited as a target for automated synthesis and code generation tools. For example, previous work on LWDF techniques has emphasized their application to DSP software implementation (e.g., through integration with C and CUDA, as presented in [112, 62]). LWDF APIs for CUDA and C have been targeted in the DIF-GPU tool for automated synthesis of hybrid CPU/GPU implementations [63].

This Chapter presents a study that deeply integrates LWDF techniques with hardware description language (HDL) programming, and that provides a complex application study involving LWDF-based digital hardware design and optimization, with emphasis on the rigorous integration of power-management within the proposed APIs.

6.1.1 LWDF

For software implementation, an LWDF actor is implemented as an abstract data type that has four interface functions, which are referred to as the *construct*, *enable*, *invoke*, and *terminate* functions. The construct and terminate functions can be viewed as an object-oriented constructor and destructor for instantiation and removal of actors, respectively. The enable and invoke functions in LWDF provide concrete mechanisms for implementing in software the corresponding functions of the same name from the abstract CFDF semantics. The LWDF enable function returns a Boolean value; the returned value is `true` if (1) there is sufficient data on the actor's input edges to execute the current mode; and (2) the output edges of the actor have sufficient empty space to accommodate the data that would be produced if the next mode were to be executed. This formulation makes sense in LWDF because of the restriction in CFDF semantics that the dataflow rate on a given port is constant for a given mode.

The invoke function of an LWDF actor A carries out a single firing of A according to its current mode; determines the next mode for a; and changes the current mode of A to be equal to this newly-determined next mode just before returning. The enable and invoke functions provide interfaces for implementing schedulers to coordinate execution of dataflow graphs that are implemented using LWDF. A broad class of schedulers can be implemented using these interfaces, including many types of static, quasi-static, and dynamic schedulers (e.g., see [89, 87]). Static schedules are generated at compile time and specify fixed sequences of actor firings, thus, the actors can be directly invoked by static schedulers without checking the firing conditions using enable functions. Quasi-static schedules are generated at compile-time, but may contain code for performing some data-dependent, scheduling-related computations at runtime. Dynamic schedulers schedule dynamic dataflow applications in ways that involve relatively large amounts of runtime decision-making.

A simple example of a dynamic scheduler is a canonical CFDF scheduler [89]. A canonical scheduler S calls the enable functions of the actors in some order in a round robin fashion. Each time the enable function of an actor A is called by S and the function returns "true", the invoke function of A is immediately executed. This process of visiting and conditionally invoking actors is repeated until no actors are enabled or some other termination condition of the application is satisfied.

As with actor design, LWDF provides a compact set of interfaces for implementing the FIFO buffers that correspond to dataflow graph edges. These interfaces provide standard functions that are used in LWDF-based actors and schedulers to work with FIFOs. Details of the implementation are unspecified so that designers have full flexibility in developing and applying different FIFOs in different applications or in different parts of the same application to achieve desired trade-offs in inter-actor communication performance (e.g., by mapping dataflow edges into different types of memories). LWDF is formulated to orthogonalize FIFO, actor, and scheduler implementation so that, for example, modifications to or replacement of a FIFO implementation do not require modifications to actors that communicate with the FIFO or scheduling logic that coordinates execution of the graph. Please

see [54] for general background on the utility of orthogonalization in system-level design. LWDF interface functions defined for FIFOs include functions for construction and termination (as with actors); reading tokens from FIFOs; writing tokens into FIFOs; and checking their token populations and amounts of available free space.

6.2 Methodology - LWDF

As discussed in Section 6.1, LWDF has primarily been targeted to DSP software implementation. This section presents the extension of the general LWDF methodology for efficient digital hardware implementation. The design techniques presented in this section are formulated concretely in the context of the Verilog HDL, referring to this integration of LWDF with Verilog as *LWDF-V*. The design concepts underlying LWDF-V can be adapted to other HDLs as they do not depend on specialized aspects within the Verilog language (i.e., aspects that do not have natural counterparts in other common HDLs).

In LWDF-V, the enable, invoke and scheduling functions for an actor are implemented as three coupled Verilog modules, which are named in this Chapter as the *actor enable module* (*AEM*), *actor invoke module* (*AIM*) and *actor scheduling module* (*ASM*), respectively. Dataflow edges are implemented as *dataflow edge modules* (*DEMs*) to provide communication channels for connections between actors. Since DEMs buffer data through a first-in first-out protocol, they are named also simply as *FIFOs*. Figure 6.1 illustrates an example of an LWDF-V actor.

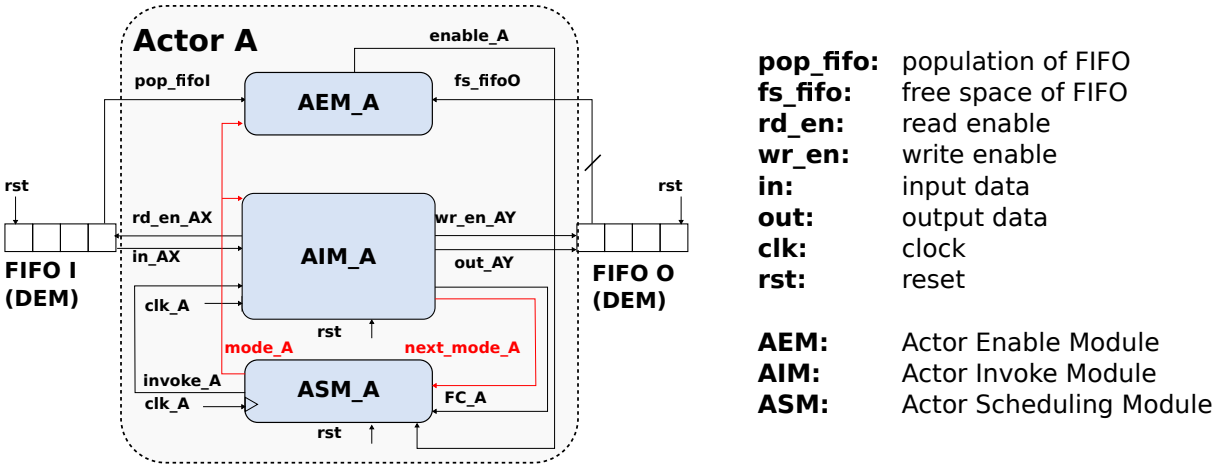


Figure 6.1: Illustration of an LWDF-V-based actor.

6.2.1 Actor Invoke Module

Recall that LWDF imposes minimal constraints on component designs. In this connection, the only requirement for an AIM is that the CFDF-based behaviour prescribed by the standard AIM operational states and interfaces (described below) is maintained. Beyond that, the AIMs can be decomposed into arbitrary hierarchies of sub-modules, and described using any Verilog coding style, including behavioural, structural, or mixed behavioural/structural coding. LWDF-V enhances the reusability and retargetability of the modules, and also facilitates evolutionary design, where sub-module designs associated with different subsystems can be progressively refined as more and more details of the targeted implementation are determined. The high level operation of the AIM is required to have two states: the *actor idle state* and *actor firing state*, which are called the *AIM operational states*

of the associated actor. The interfacing requirements of AIMs are defined in terms of these two states. The required interface ports for the AIM are divided into the following four groups.

- **Dataflow-related input ports**: This group of ports contains one input port corresponding to each input port X of the associated CFDF actor A, and a Boolean input port called invoke to initiate the next firing of the actor on the next clock cycle if the actor is currently in the idle state. In Figure 6.1, examples of signals sent to ports within this group include in_AX and invoke_A .
- **Dataflow-related output ports**: This group of ports contains one output port corresponding to each actor output port Y, one Boolean write enable port corresponding to each output port for submitting write requests to the output edge, and one Boolean read enable port corresponding to each input port X for submitting read requests to the input edge. Examples in Figure 6.1 of signals sent from ports in this group include out_AY, wr_en_AY, and rd_en_AX.
- **Platform-related input ports**: This group of ports contains a clock input port for relevant synchronization with interfacing circuitry, and a synchronous Boolean reset input port to bring the actor to its idle state on the next clock cycle. Examples in Figure 6.1 of signals sent to ports in this group are clk_A and rst_A.
- **Control-related input ports**: For a given actor A, this group contains a port called mode that provides the current CFDF mode of the actor. In Figure 6.1, examples of signals sent to this port are mode_A and FC_A .
- **Control-related output ports**: For a given actor A, this group contains a port called next_mode that provides the next possible CFDF mode of the actor, and a port called FC, which stands for *firing complete*, that sends a Boolean signal to indicate when a firing of A completes during the current clock cycle. In Figure 6.1, examples of signals sent from these ports include next_mode_A and FC_A .

Figure 6.2 provides an example of the FSM control flow for an AIM with three modes. The AIM stays in the current mode x until the firing related to the mode is completed. When this firing is completed, the FC signal is driven high, and the next_mode is suggested. In each mode, a sub-FSM controls the execution; the AIM waits in an IDLE state until invoke is high, then the state is updated to FIRING,where the AIM consumes input data, executes the actor operation and produces output data according to the current mode.

6.2.2 Actor Enable Module

To provide a standard interface for the CFDF-based enable function described in Section 6.1.1, the AEM module for an LWDF-V actor contains the following required interface ports.

- **Population and free space ports**: for each input port X of the associated CFDF actor A, the AEM has one input port for accessing information about the buffer state. This port provides the current buffer population (the number of tokens in the input buffer) for the FIFO I that is connected to port X. Similarly, for each output FIFO O that is connected to an output port Y, the AEM has one input port that provides the free space level (the output buffer capacity minus the population). These input ports are named, in terms of the associated FIFO names, as pop_fifoI and fs_fifoO, respectively.

  Figure 6.3 shows two examples of inter-actor communication. In the first example, two actors A and B exchange data through two FIFOs. Actor A has three ports to access the buffers'

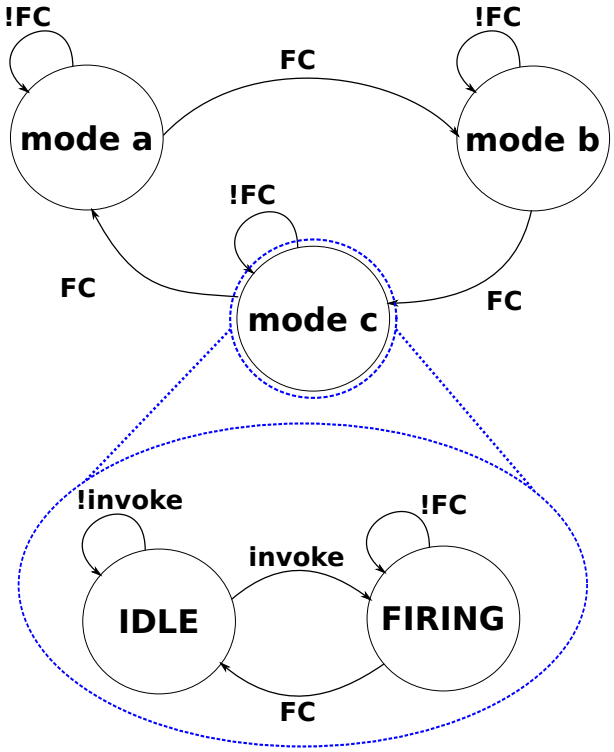


Figure 6.2: Example of an AIM FSM for a CFDF actor with three modes.

state: `pop_fifo1` provides the population of `FIF01` connected to `input 1`, and `fs_fifo2` and `fs_fifo3` provide the free space levels of `FIF02` and `FIF03` connected to output ports `output 1` and `output 2`. In the second example, an actor `C` sends output data to two actors `D` and `E` through two FIFO channels, `FIF05` and `FIF06`. For each output FIFO `C` has one free space input signal — these signals are labelled as `fs_fifo5` and `fs_fifo6`.

- **Mode port**: This input port is used to specify the CFDF mode (for the enclosing actor `A`) relative to which the AEM will perform its next fireability testing operation (i.e., its operation to determine whether or not there is sufficient data and free space available to permit a firing of the actor). This port is named in terms of the enclosing actor `A` as `mode_A`. Figure 6.4 shows an example with three different firing conditions $\{C_x\}$ for three different modes `mode x`. When the firing condition is `true`, the `enable` signal is set high.

- **Enable port**: This output port is driven with the Boolean result produced by checking the fireability of the enclosing actor in the mode specified by the AEM mode port. This port is named in terms of the enclosing actor `A` as `enable_A`.

The AEM can be implemented using combinational or sequential logic. The latter form may be preferred, for example, if large numbers of ports are involved and it is desired to share hardware resources across the comparison operations that are involved in the fireability checking process. However, for many practical actors and implementation scenarios, the number of inputs is relatively small and combinational AEM realization is a reasonable design choice. In the work only the combinational AEM implementation is considered.

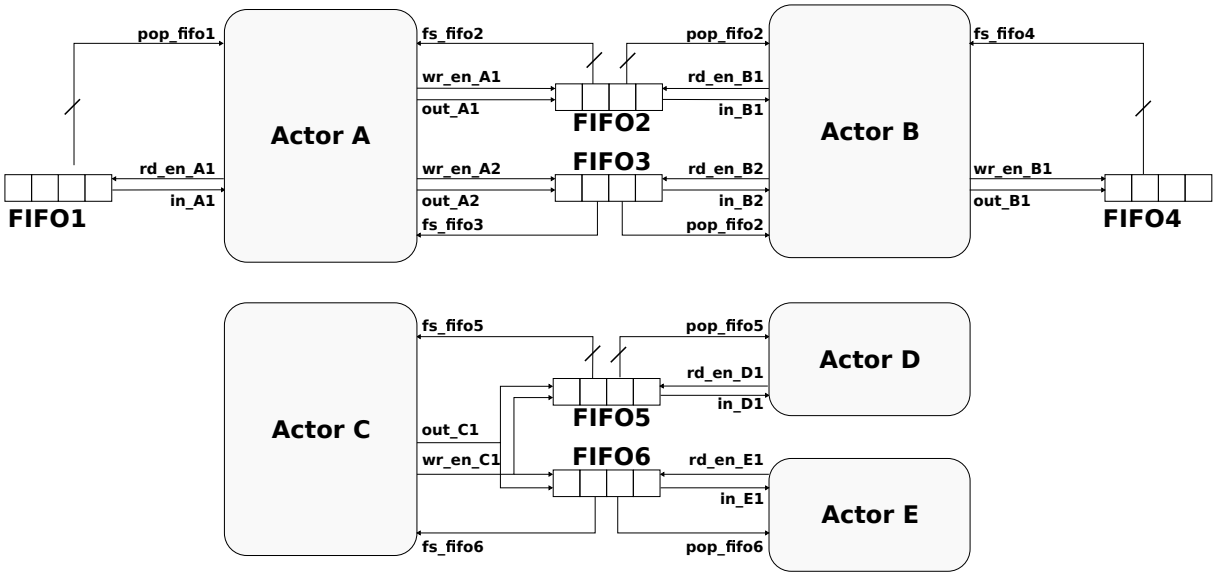


Figure 6.3: Illustration of LWDF-V-based actors communication.

Mode	Condition
mode a	$\mathbf{C_a}$ = (pop_fifo1 >= consumption_rate_of_a)&&(fs_fifo2 >= production_rate_of_a)
mode b	$\mathbf{C_b}$ = (pop_fifo1 >= consumption_rate_of_b)&&(fs_fifo2 >= production_rate_of_b)
mode c	$\mathbf{C_c}$ = (pop_fifo1 >= consumption_rate_of_c)&&(fs_fifo2 >= production_rate_of_c)

condition	enable
true	1
false	0

Figure 6.4: Example of a AEM with three different firing condition for three possible modes.

6.2.3 Actor Scheduling Module

The ASM is an actor-level subsystem that determines the next mode of the associated actor after the actor firing is completed, and invokes the actor firing after the actor is enabled again. Compared to schedulers discussed in Section 6.1.1, which control groups of actors (i.e., related to specific design subsystems or to the entire digital system that is being developed), the ASM can be used to implement a *fully distributed* scheduling approach. In such an approach, an actor is scheduled to begin a new firing whenever it is idle and its enable condition is satisfied. Use of the ASM can also be mixed with the kinds of schedulers discussed in Section 6.1.1. Using such hybrid strategies, selected actors can be scheduled in a distributed fashion (using ASMs), while the execution of other actors is coordinated using centralized mechanisms. The ASM supports one specific scheduling strategy that can be used to implement LIDE-V systems. Its use is not required; modules that implement other kinds of scheduling strategies can be used instead or in combination, as described above.

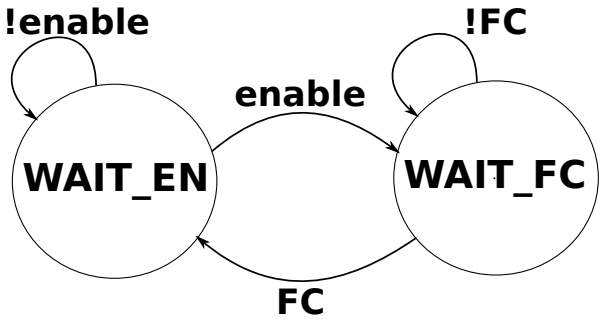
The interface ports of an ASM are listed as follows. For an ASM that is associated with a given actor *A*, these ports are described here in relationship to the AIM and AEM of the same actor *A*.

- Dataflow-related input ports: This group of ports contains a Boolean input port that is connected to the output port FC of the AIM, a Boolean input port that is connected to the output

port enable of the AEM, and an input port that is connected to the output port next_mode of the AIM.

- Platform-related input ports: This group of ports is the same as the platform-related input ports introduced in Section 6.2.1.
- Control-related output ports: This group contains an output port called mode and an output port that is connected to the input port invoke of the AIM. The ASM makes the decision on the current actor mode and sends the resulting mode signal to the AEM and AIM through the output port mode. Thus, the ASM is responsible for carrying out mode transitions of the actor after firings are completed. Generally, the ASM can either set the actor mode to be the next mode signal received from the next_mode input port or ignore the next mode signal and set the mode according to some user-defined logic that is integrated as part of the ASM. The ASM also launches the next actor firing once the previous firing is completed and the actor is enabled again.

Figure 6.5 shows an example of an FSM that controls an ASM. The ASM waits in the state WAIT_EN for the enable signal to become high. When enable = 1, the ASM invokes the AIM (invoke = 1). Then the ASM waits in the state WAIT_FC for the firing completion (FC = 1), and sets the next actor mode.



state	enable	invoke
WAIT_EN	1	1
WAIT_EN	0	0
WAIT_FC	x	0

state	FC	mode
WAIT_EN	x	current_mode
WAIT_FC	0	current_mode
WAIT_FC	1	next_mode

Figure 6.5: An example of an FSM that controls an ASM.

The example in Figure 6.6 illustrates temporal relationships among the enable, invoke and FC signals. After the enable signal is high, the ASM raises the invoke signal. The AIM then executes its operation and the ASM waits for the firing completion signal FC. T_{ei} is the elapsed time between the instant when the actor becomes enabled, and when the corresponding firing is invoked. Similarly, T_{ic} is the invocation to firing completion time, and T_{ci} is the time between the firing completion for one invocation and the start of the next invocation. Finally, T_{ec} is the enable to firing completion time, and T_{ii} is the elapsed time between two successive invocations. Such measurements can provide insight into the performance of the given actor in the context of the applied scheduling strategy.

6.2.4 Dataflow Edge Module

The DEM is used in LWDF-V to implement a dataflow edge. The required interface ports include the following.

- **Enable ports**: These two Boolean input ports provide read enable and write enable signals, rd_en and wr_en, for accessing the FIFO storage.

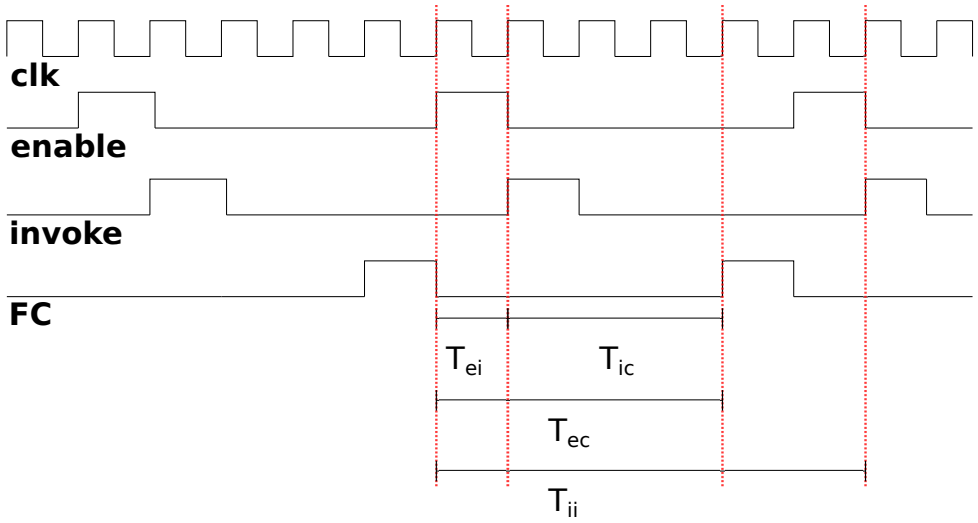


Figure 6.6: Examples of signal waveforms during execution of an LWDF-V actor.

- **Data input/output ports**: These ports, named `in` and `out`, are used by the FIFO to read input data and send output data, respectively, when a read or write operation is initiated.
- **Population port**: This output signal, named `pop`, is driven with a non-negative integer value that gives the number of tokens that is currently stored in the FIFO.
- **Free space port**: This output signal, `fs`, provides the current value of $(c-p)$, where c and p are the capacity and population, respectively, of the buffer.

Figure 6.7 depicts an overview of an LWDF-V-based synchronous FIFO design. Once the FIFO read or write operation is enabled, the `rd_addr` or `wr_addr` module will update the read or write pointer accordingly. The population `pop` and free space `fs` signals will be updated as well.

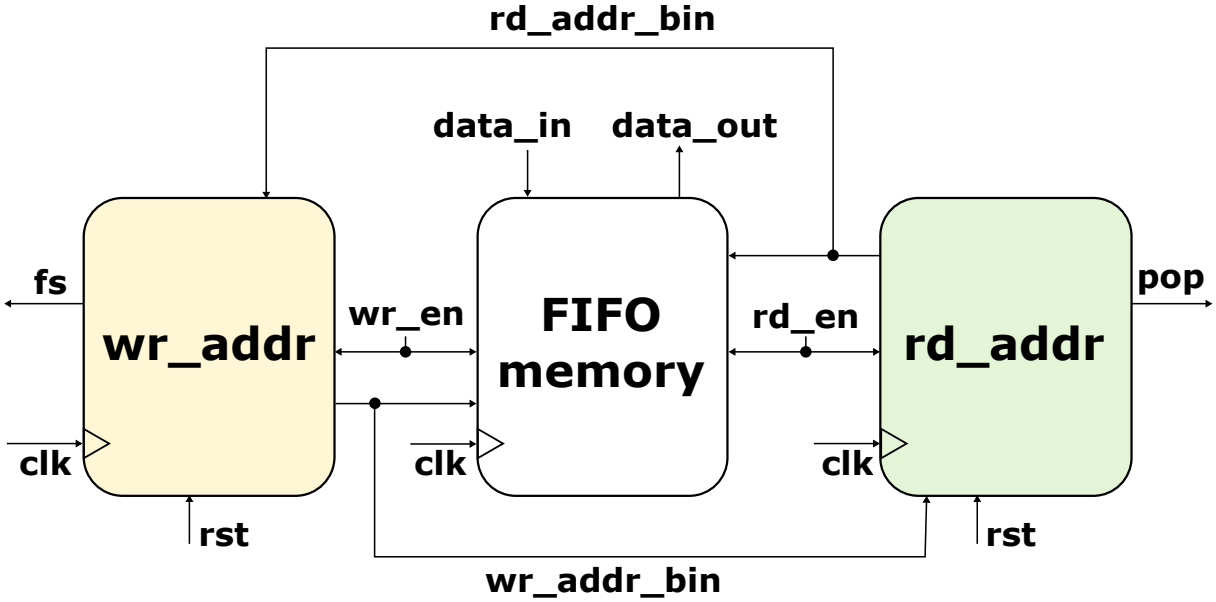


Figure 6.7: Synchronous FIFO design

6.3 Lightweight Dataflow Environment for LWDF-V methodology - LIDE-V

The *lightweight dataflow environment* (*LIDE*) provides implementations of APIs, graph element (actor and edge) libraries, and other utilities that support the LWDF programming methodology (see [103, 102]). This section presents an extension of LIDE called LIDE-V that provides concrete support for the abstract LWDF-V methodology. LIDE-V presents capabilities that help to address challenges brought about by the need to handle actors with arbitrary variations in complexity, and it also discusses the relevance of these capabilities to low power signal processing.

In particular, this section presents an asynchronous and GALS-oriented design methodology using LIDE-V for heterogeneous-complexity and low power implementation. Specifically, the proposed methodology applies (1) asynchronous communication between actors that utilizes multiple clock domains, where the bottleneck actors are executed at higher clock frequencies and the others at lower frequencies; and (2) a clock gating technique that "switches off" idle actors to reduce dynamic power consumption. As part of LIDE-V, two DEM implementations — one for synchronous FIFO realization and the other for asynchronous realization — are presented. These alternative implementations provide important examples of support for orthogonality in LWDF-V, and its use to integrate alternative actor and edge implementations.

The novelty of this development centres on the systematic integration of asynchronous design, GALS, and clock gating techniques with lightweight dataflow programming interfaces and their underlying CFDF model of computation. This integration with CFDF is notable in turn due to the utility of CFDF as a foundation for working with various specialized and heterogeneous forms of dataflow (e.g., see [89]). Furthermore, the orthogonality among CFDF components lays a valuable foundation for asynchronous design, and the proposed clock gating techniques exploit the enable/invoke semantics in CFDF.

6.3.1 Asynchronous LIDE-V Design

A *dataflow clock domain* (or simply "clock domain" when the dataflow context is understood from context) can be defined as a maximal set of actors that is driven by the same clock signal. In asynchronous LIDE-V designs, different parts of a dataflow graph can be driven by different clock signals, thus forming multiple clock domains. This in turn allows "slower" actors to be placed in higher frequency clock domains, so that they can be accelerated without having to increase the power consumption of the whole design linearly with the clock frequency, and "faster" actors to be placed in relatively low frequency clock domains, so that the downtime between "faster" and "slower" actors can be reduced.

Figure 6.8 depicts an example of a LIDE-V design with two clock domains, where actor `A` is driven by `clk_1`, and actors `B` and `C` are driven by `clk_2`. Communication channels between actors in the same clock domain are called *synchronous FIFOs*. These FIFOs outline the "synchronous islands" within the overall GALS design. The problem of passing data between synchronous islands is addressed using the asynchronous DEM, named also *clock domain crossing* (*CDC*) FIFOs. In Figure 6.8 `FIF03` is a *synchronous FIFO* while `FIF02` is a *CDC-FIFO*.

Figure 6.9 illustrates the CDC FIFO design adopted in LIDE-V, which is based on Cummings's design presented in [27]. CDC FIFOs are driven by two different clock signals, one for read operations (`rd_clk`), and another for write operations (`wr_clk`). The LIDE-V CDC FIFO presents some adaptations to Cummings's design so that it is consistent with the LIDE-V framework. For example, Cummings's design only provides the `empty` and `full` signals, calculated inside the `wr_addr` and `rd_addr` modules. In order to generate the required population `pop` and free space `fs` signals, the `wr_addr` and

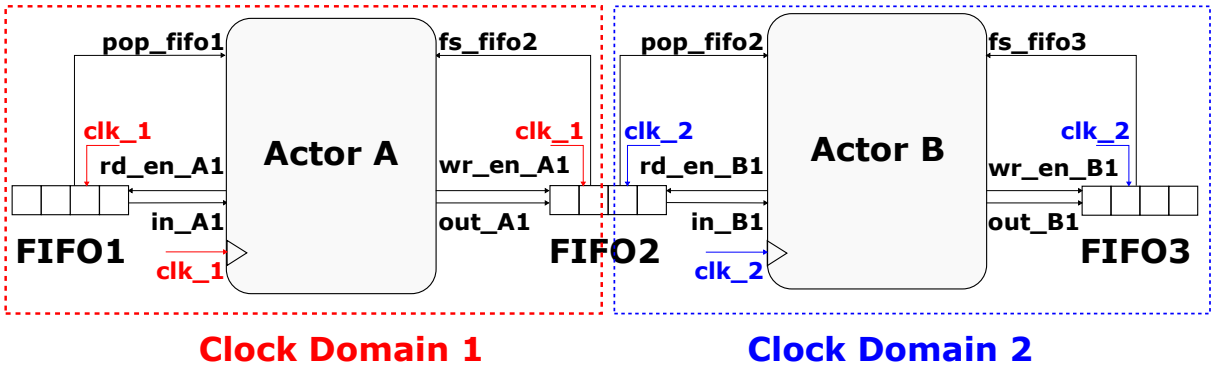


Figure 6.8: Illustration of an LWDF-V-based implementation of a CFDF graph that consists of three actors.

rd_addr modules are modified to calculate these signals by computing the offset between the write pointer and read pointer.

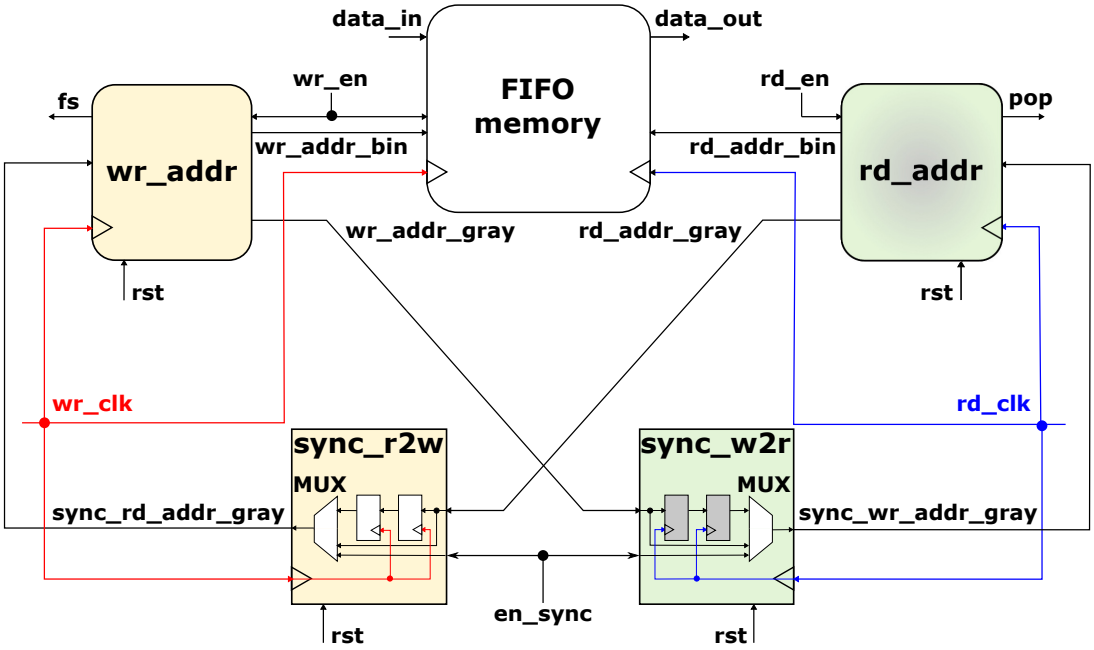


Figure 6.9: Asynchronous FIFO design in LIDE-V.

To support clock gating within FIFOs, the LIDE-V CDC FIFO presents another modification compared to Cummings's design. The objective here is to allow all of the logic units belonging to the corresponding clock domain to be turned off when rd_clk or wr_clk is off. In Figure 6.9, for example, the modules in gray are disabled when the rd_clk signal is off. This behaviour is not considered in Cummings's design and if the synchronization circuits are off, in that design, the read and write pointers would not be sent to the wr_addr and rd_addr modules. In order to guarantee that the updates of population pop and free space fs are performed properly, the syn_r2w and sync_w2r modules both have multiplexers that are responsible for sending read and write pointers, respectively, when one or both of the two clocks is disabled.

6.3.2 Clock Gating

In LIDE-V, clock gating is applied at the actor level to switch off idle components in the dataflow graph. Figure 6.10 depicts an overview of a clock-gated LIDE-V design. Actor-level clock gating is achieved systematically as a natural by-product of the LIDE-V design technique, which in turn allows designers to apply clock-gating more thoroughly and more reliably. Clock gating is applied to a LIDE-V actor by adding a clock gating module `CGM_A` to the original LIDE-V actor design illustrated in Figure 6.1.

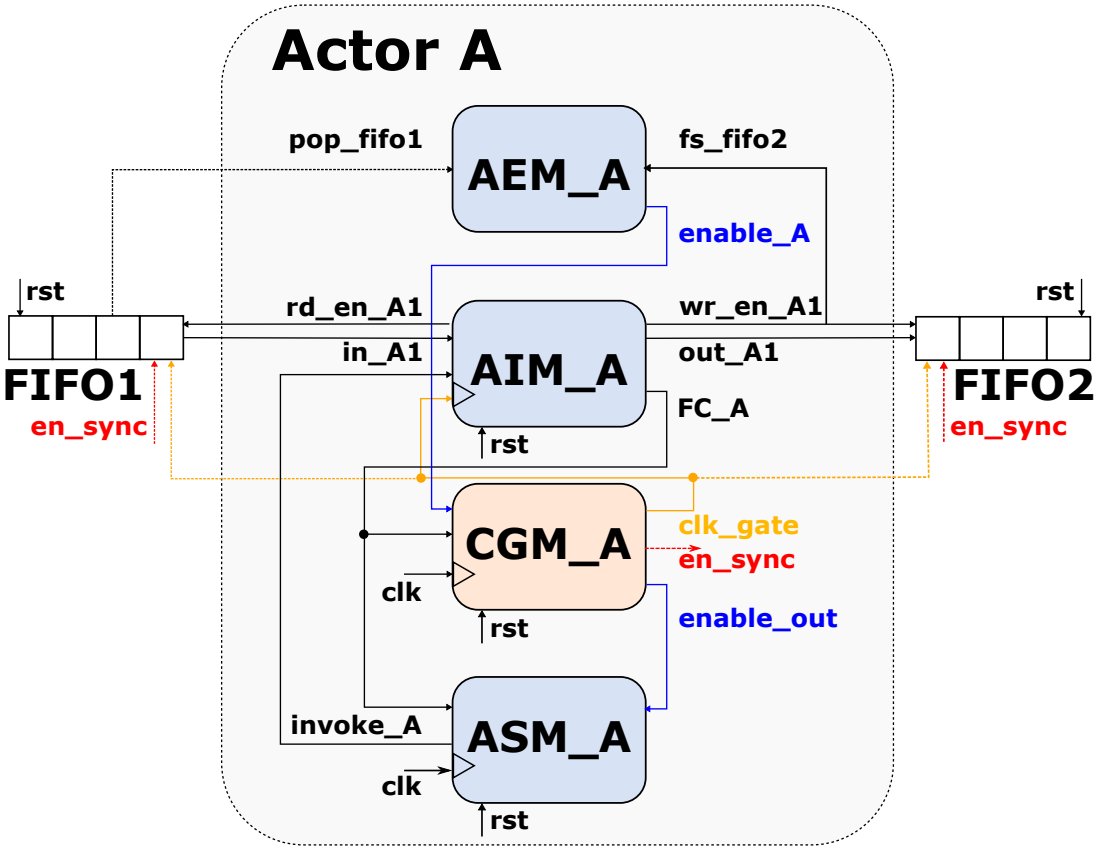


Figure 6.10: Clock gating in a LIDE-V actor.

The LIDE-V-based clock gating technique exploits the graph execution information provided by the enable `enable_A` and the firing complete `FC_A` signals. Figure 6.11 illustrates how the signals are related: when `enable_A` from the `AEM_A` is high, `CGM_A` enables the clock (`en_clk` is high), and switches the clock off when the actor has finished its computation — i.e., when the `FC_A` signal from `AIM_A` is high.

From a technical point of view, the clock is disabled in two different ways, depending on the target technology. In ASIC designs, it is possible to modify the clock by exploiting some custom logics (e.g., a simple *AND* gate can be used to disable the clock signal), while in FPGAs it is necessary to use dedicated blocks (*BUFGCEs*) to switch it off. The `CGM_A` also delays the `enable_A` signal by two clock cycles so that, after an OFF-to-ON transition, the AIM has enough time to be active before it receives the `invoke_A` signal from the scheduler.

Clock Gating of Dataflow Edge Modules in Asynchronous Designs

The clock gating technique introduced above can be applied to both synchronous and asynchronous designs. Moreover, in asynchronous designs, the clock gating technique can be applied not only to

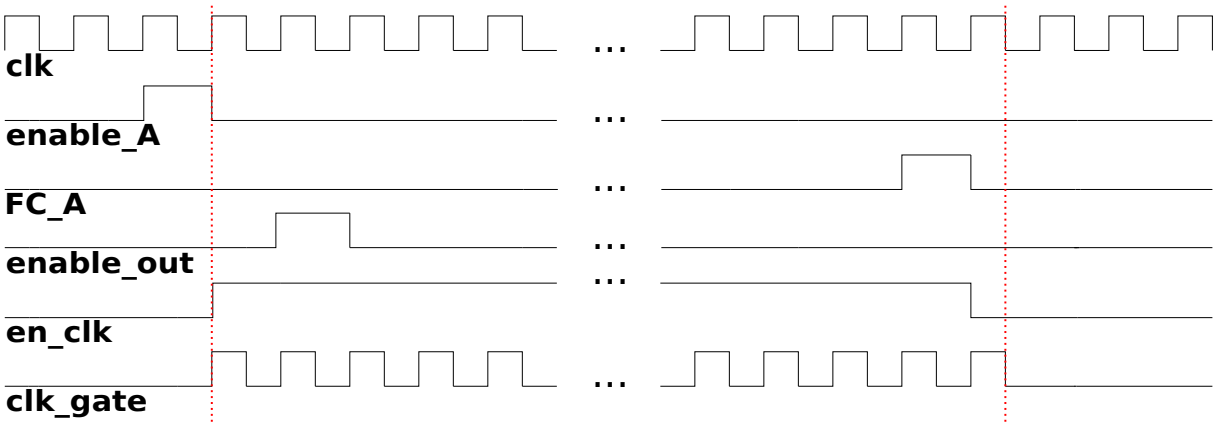


Figure 6.11: Signal waveforms in the clock gating module.

actor modules but also to the CDC FIFO modules, as mentioned in Section 6.3.1. When an actor is idle, it does not read/write data from/to its input/output FIFOs, so the read clock of its input FIFOs and the write clock of its output FIFOs can be disabled to save even more power. Then the CDC FIFO will turn off the corresponding logic units as mentioned in Section 6.3.1.

To ensure correct updates of the output signals pop and `fs`, one additional signal called `en_sync` is sent from the `CGM` to the CDC FIFO module indicating that either one or both of the `rd_clk` and `wr_clk` is/are disabled .

Clock Gating of Dataflow Edge Modules in Synchronous Designs

In synchronous designs, the clock gating technique cannot be applied to the synchronous FIFO design introduced in Section 6.2.4 because the modules that update the signals related to read and write operations are driven by the same clock signal.

One way to enable clock gating of DEMs in synchronous designs is to replace the synchronous FIFOs with the CDC FIFOs mentioned in Section 6.3.2. However, compared with synchronous FIFOs, CDC FIFOs require more hardware resources and consume more power. Thus, the power saved by switching off the unused logic units may be counteracted by the power overhead introduced by the additional resources. This section presents a new FIFO, where the `wr_addr` and `rd_addr` blocks are synchronized to different clock signals, and a dual-clock FIFO memory is developed. This new FIFO design, which we call a *pseudo-CDC FIFO*, is illustrated in Figure 6.12. Compared with the CDC FIFO design, the pseudo-CDC FIFO design does not contain synchronization or gray coding circuits, since the `wr_clk` and `rd_clk` clock signals are always connected to the main clock. Similar to CDC FIFOs, the unused logic units in the pseudo-CDC FIFOs can be turned off by clock gating technique to save more power.

6.3.3 FIFOs comparison

Table 6.2 compares the resource utilization of the synchronous, asynchronous (CDC) and pseudo-asynchronous (pseudo-CDC) FIFO designs presented in this paper. The data is extracted from the post-implementation reports of the four FIFOs, all with the capacity being 768 and bit-width being 64.

As it is possible to see from Table 6.2, the CDC FIFO requires the most resources compared to the other FIFO designs. This is due to the additional synchronization and gray coding circuits in the CDC FIFO. Additionally, the pseudo-CDC FIFO has the same resource utilization as the synchronous FIFO, but is adapted to support clock gating in synchronous designs.

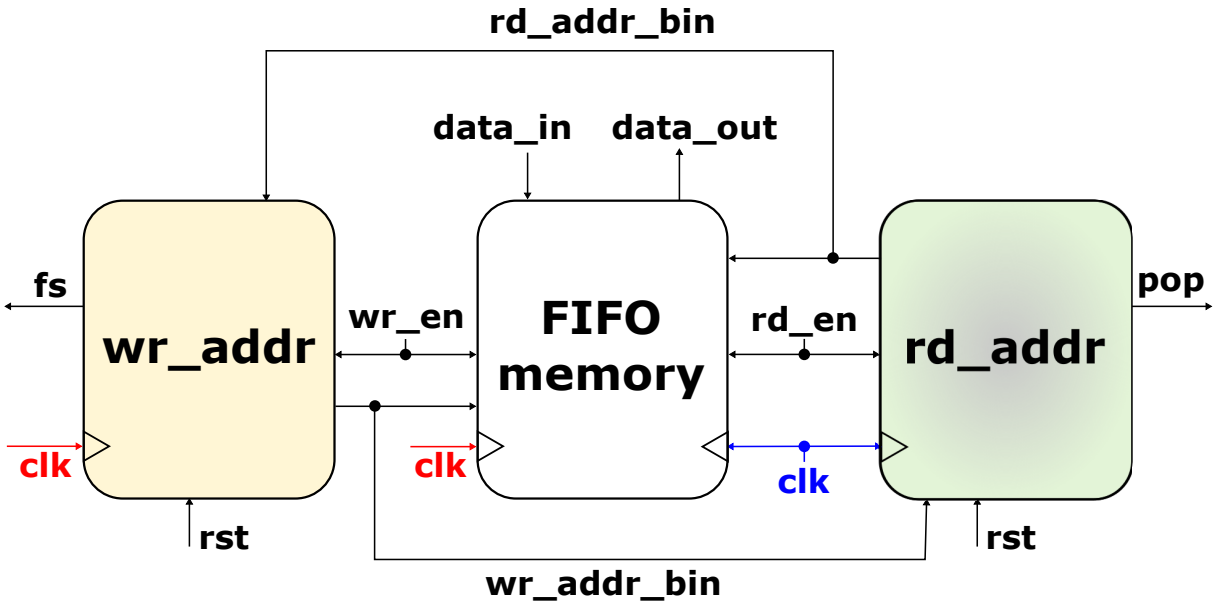


Figure 6.12: Pseudo-CDC FIFO design in LIDE-V.

Table 6.1: Resource utilization. The numbers in parentheses give the differences in utilization of the corresponding resource compared to the synchronous FIFO.

	LUTs	%	*REGs*	%	*BRAMs*
Synch.\FIFO	73	–	22	–	2
CDC FIFO	199	+172.60	88	+300	2
Pseudo-CDC FIFO	73	+0	22	+0	2

Table 6.2: Resource utilization of the implemented FIFOs.

	FIFO Type	*LUTs*	*REGs*	*BRAMs*
FIFO	Synch.	73	22	2
CDC FIFO	Asynch.	199	88	2
Pseudo-CDC FIFO	Pseudo-Asynch.	73	22	2

6.4 Experimental Results

This section demonstrates the capabilities of LIDE-V through a deep neural network (DNN) application. The DNN system employed in this case study, based on a neural network presented in [48], is designed to automatically classify vehicles into four types — bus, truck, van and car. Details on algorithmic aspects of this DNN are reported in [48].

The DNN architecture, which is summarized in Table 6.3, consists of five layers: two convolutional layers, CONV1 and CONV2, followed by two dense layers, DENSE1 and DENSE2, plus an output layer, OUTPUT. The first convolutional layer maps red, green, and blue (RGB) channels of an input image of size 96×96 into 32 separate 48×48 feature maps. The second convolutional layer re-maps the 32 feature maps generated by the first layer into 32 24×24 feature maps. The third and fourth layers are fully connected layers with 100 feature nodes in each layer. The results of the two fully connected dense layers are fed to the output layer, which returns a vector with four elements, each representing the probability that the vehicle in the input image belongs to one of the four vehicle types.

The computational complexity and amount of required data transfer for each layer are summa-

Table 6.3: DNN hyperparameters.

Hyperparameter	**Selected Value**
Number of Convolutional Layers	2
Number of Dense Layers	2
Input Image Size	96×96
Kernel Size in All Convolutional Layers	5×5
Number of Feature Maps	32

rized in Table 6.4. From this table, it is possible to notice that the convolutional layers require much more computation while requiring significantly less data transfer compared with the other layers. The first convolutional layer is the one selected for hardware acceleration. The associated dataflow subgraph takes as input a $w \times h$ RGB image and computes a single $(w/2 \times h/2)$ feature map as output, taking three $a \times b$ convolution kernels as parameters. The whole layer, which outputs 32 feature maps, can then be accelerated by firing the graph repeatedly with different kernels.

Table 6.4: The computational complexity and amount of data transfer for each DNN layer.

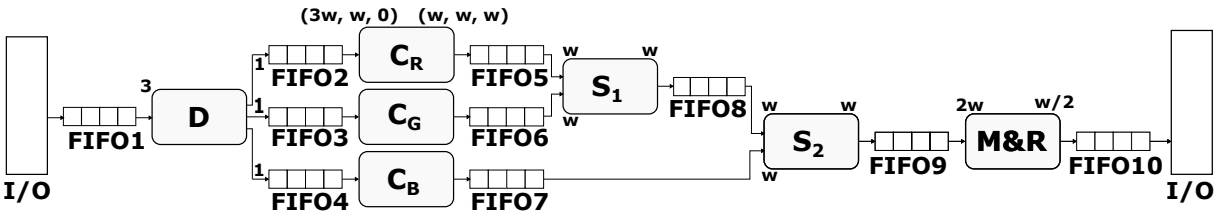
Layer	Multiplication	Addition	Data transfer (in double precision format)
CONV1	22,118,400	23,003,136	103,776
CONV2	58,982,400	61,341,696	117,760
DENSE1	1,843,200	1,843,200	1,868,900
DENSE2	10,000	10,000	10,200
OUTPUT	400	400	504

6.4.1 LWDF-V Implementation of Deep Learning Neural Network Application

Figure 6.13 shows an overview of the DNN dataflow subgraph implemented using LIDE-V. The annotations next to the actor ports represent the associated production and consumption rates. The convolution actors, *conv_r*, *conv_b* and *conv_g*, each have three CFDF modes with different data rates among the modes. The symbol w represents the image width.

In Figure 6.13, the *Deinterleave* actor (D) separates the RGB values of an input image to generate three corresponding images. Then the *Convolution* actors perform matrix convolution on each of these single-color images to generate a set of convolved images with the same size as the input images (C_R, C_G and C_B). These convolved images are then added together by the *Sum* actors (S_1 and S_2). Finally, the feature map is downsampled to 48×48 resolution by the *Maxpool&rRelu* actor ($M\&R$). To achieve this downsampling, $M\&R$ partitions the map into a set of sub-regions; for each sub-region, the actor selects the maximum element as the representative of the sub-region, and clamps the negative values among the selected ones to zero to generate the final output feature map.

By applying the modularized decomposition of CFDF semantics in LWDF-V, each actor of the DNN subgraph has been implemented as three coupled Verilog modules (AEM, AIM and ASM); actors communicate through the DEMs. The DEM is parameterized with information about its dimension; by changing the parameter appropriately, it is possible reuse the same module across the subgraph. Also, the AEM is parameterized with information about actor modes and production/consumption rates, and it can be reused across all actors. Furthermore, this case study uses a static scheduling



D: Deinterleave
C_R: Convolutional Red
C_G: Convolutional Green
C_B: Convolutional Blue
S_1: Sum 1
S_2: Sum 2
M&R: Maxpool&Relu

Figure 6.13: LIDE-V design for the accelerated DNN subgraph.

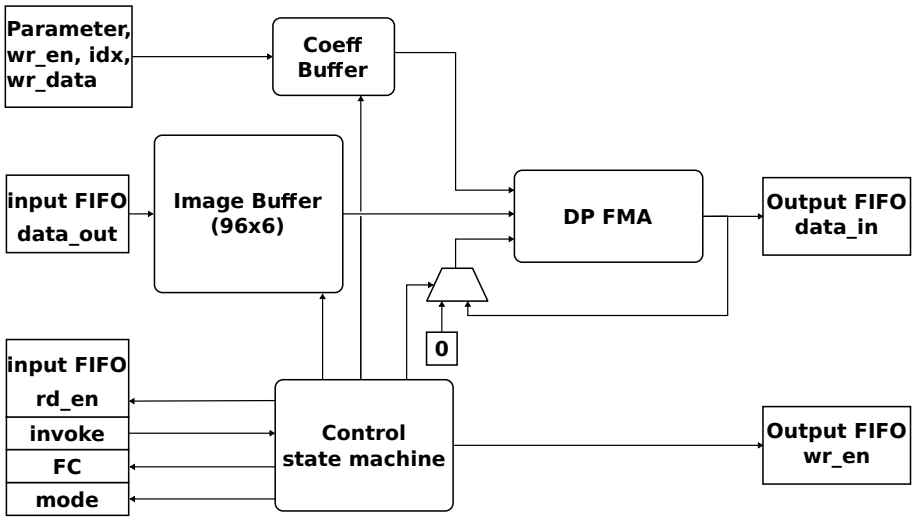


Figure 6.14: The AIM implementation for the convolution actor.

technique, and the ASM design is reused for all actors. In the presented design, it is necessary to develop develop customized implementations only of the AIM for the different actors in the DNN accelerator subsystem.

As an example of AIM implementation in the DNN accelerator, Figure 6.14 illustrates the structure of the AIM for the convolution actor. The DNN accelerator applies three instances of this actor. The major components of this actor include an image buffer that is large enough to hold six rows of image data, a kernel coefficient memory, and a floating point fused multiplier-adder (FMA). When initialized, the actor settles in the first mode, which consumes 3 rows and convolves them with the kernel to produce a single row of output. Then, it iterates through the image row by row in the second mode producing and consuming 1 row per firing. Finally, the actor transitions to the third mode and produces the last row of the output image without consuming any input tokens.

6.4.2 Hardware Profiling

This section presents the implementation of the DNN application introduced in Section 6.4.1 using the Xilinx Zynq-7 ZC706 platform and execution time data for each type of actor. This hardware profiling data provides further insight about selection among the different low power techniques mentioned in Section 6.3.

Table 6.5 depicts the execution time (in clock cycles) of the DNN subgraph, illustrated in Figure 6.13, to generate one of the 32 feature maps (*DNN_1fm*), and the average execution time of each type of actor. Here, each symbol of the form Z_{xy} represents the averaged value of T_{xy} across the entire process of generating *DNN_1fm* (see Section 6.2.3 for definitions of the T_{xy} symbols). For example, Z_{ei} represents the averaged value of T_{ei}.

Additionally, *#firings* indicates how many times an actor is fired during the generation of *DNN_1fm*. TA_T_{ic} is the total execution time of an actor, and is equal to $(Z_{ic}) \times (\#firings)$. $TA\_T_{ic}\%$ is the ratio (expressed as a percentage) TA_T_{ic}/t_{total}, where t_{total} is the total time required to generate *DNN_1fm*. Figure 6.6, which has been drawn considering the `enable`, `invoke` and `FC` signals of the *Deinterleave* actor, gives insight into how these quantities and their associated signals are related.

According to the $TA\_T_{ic}\%$ data derived from this experiment, the *Convolution* actors are active during 99.04% of the total execution time, thus they do not benefit from clock gating application. On the other hand, there is potential to save significant amounts of power by applying clock gating to the *Deinterleave*, *Sum* and *Maxpool&Relu* actors.

Table 6.5: Execution time in clock cycles of each actor. t_{total}: total time required to generate *DNN_1fm*. Z_{ic}: invoke to firing completion time. Z_{ei}: enable to invoke time. Z_{ec}: enable to firing completion time. $\#firings$: number of firings during generation of *DNN_1fm*. $TA\_T_{ic} = (Z_{ic}) \times (\#firings)$. $TA\_T_{ic}\% = (TA\_T_{ic}/t_{total}) \times 100$.

	t_{total}	Z_{ic}	Z_{ei}	Z_{ec}	$\#firings$	TA_T_{ic}	$TA\_T_{ic}\%$
DNN_1fm	232,831	—	—	—	—	—	—
Deinterleave	—	3	1	4	9216	27,648	11.87
Convolution	—	2402	1	2403	96	230,592	99.04
Sum	—	107	1	108	96	10,272	4.41
Maxpool&relu	—	195	1	196	48	9360	4.02

Table 6.6 presents more data related to actor waiting times that is derived from hardware profiling. Again, Z_{xy} values are averaged versions of the corresponding T_{xy} values, as described above. The ratio between Z_{ii} and Z_{ic} measures the extent of actor idleness, which helps in gaining more insight into the multiple clock domain formation and the selection of the clock frequency for each clock domain. Here, it is possible to see that the *Deinterleave* and *Convolution* actors have much smaller levels of idleness — both wait only two clock cycles between firing completion and the next invocation (Z_{ci}). Thus, D, C_R, C_G and C_B can be grouped in a region that works at a high clock frequency. *Sum* and *Maxpool&Relu* actors have much larger values of Z_{ci}, thus S_1, S_2 and $M\&R$ can be placed in a region that works at a slower clock frequency. In particular, the data in this table on Z_{ii}/Z_{ic} suggests that the slower clock could be around 20 times smaller than the faster clock.

Figure 6.15 illustrates the two different clock regions of the DNN subgraph, identified according to analysis of Table 6.6. *Region 1* contains D, C_R, C_G and C_B that can work at a higher clock frequency, while *Region 2* contains S_1, S_2 and $M\&R$ that can work at lower clock frequency. In such an asynchronous design, *FIFO5*, *FIFO6*, *FIFO7* and *FIFO10* are CDC FIFOs as described in Section 6.3.1

Table 6.6: Waiting time in clock cycles for each actor. Z_{ic}: invoke to firing completion time. Z_{ci}: firing completion to next invoke. $Z_{ii} = Z_{ic} + Z_{ci}$.

	Z_{ic}	Z_{ci}	Z_{ii}	Z_{ii}/Z_{ic}
Deinterleave	3	2	5	1.67
Convolution	2402	2	2404	1.08
Sum	107	2297	2404	22.47
Maxpool_relu	195	4613	4808	24.65

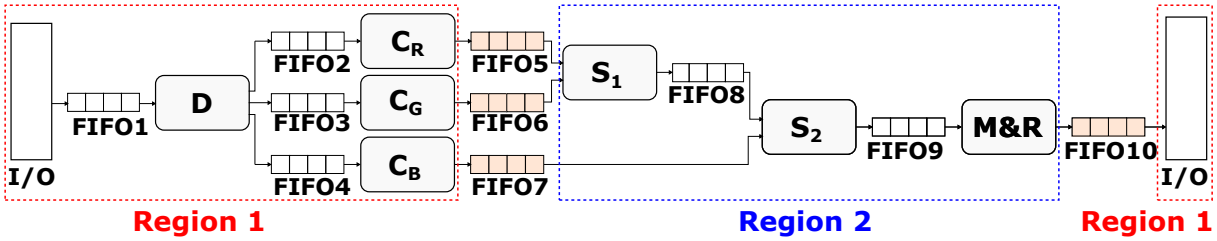


Figure 6.15: Clock Regions in DNN subgraph.

6.4.3 The Application of Low Power Techniques

Based on the hardware profiling data of the DNN application presented in Section 6.4.2, This Section presents four alternative implementation for the LWDF-V-based DNN subgraph, where the asynchronous and low power design techniques for LIDE-V described in Section 6.3 are adopted. These designs can be viewed as different ways of implementing the structure illustrated in Figure 6.15 depending on the specific DEMs that are instantiated as well as whether or not clock gating or multiple clock domains are applied. The four implementations are described as follows.

- **DNN_a**: an asynchronous DNN design. According to profiling data in Table 6.6 it is possible to allocate two clock domains: a fast clock domain cor *Clock Domain 1*, for the I/O interface, D, C_R, C_G and C_B, and a slower clock domain, *Clock Domain 2*, which has a 20 times slower clock signal, for the remaining actors. *Clock Domain 1* corresponds to *Region 1* in Figure 6.15, and *Clock Domain 2* corresponds to *Region 2*. Actors in different clock domains are connected by CDC FIFOs.

- **DNN_cg**: the baseline DNN design with integration of clock gating. According to data depicted in Table 6.5 all of the actors take advantage of the clock gating application except the convolution actors. From the profiling experiments, it is clear that the convolution actors are active for 99.04% of the total execution time, while the other actors are active for maximum 11.91% of total execution time. Since the convolution actors are operating almost continuously, there would be little power saved by clock gating them in exchange for the extra logic cost that must be incurred to implement clock gating strategy.

- **DNN_acg**: the asynchronous DNN design with integration of clock gating. Here, the asynchronous design is like the in DNN_a. However, in this design, clock gating is also applied. In particular, clock gating is applied to all of the actors except the convolution actors (as in DNN_cg), and clock gating is also applied to the CDC FIFOs.

- **DNN_facg**: the DNN design with integration of clock gating, where all the FIFOs are replaced with pseudo-CDC FIFOs mentioned in Section 6.3.2. Clock gating is applied to all of the actors except the convolution actors (as in DNN_cg) and to all of the pseudo-CDC FIFOs as well.

In addition to the above described designs, there is a further design:

- **DNN_auto**: this is the baseline *DNN* design, synthesized and implemented by enabling the automatic power optimization performed by the Xilinx Vivado tool. When this option is enabled, during the implementation step Vivado looks at the output logic of sourcing registers that do not contribute to the result for each clock cycle and then creates fine-grained clock gating and/or logic gating signals that neutralize unnecessary switching activity.

Table 6.7 summarizes the composition of the baseline DNN design and the four designs described above, reporting the number of used clocks, clock-gated (CG) actors, the actors belonging to the fast and slow clock domains, and the number of the synchronous FIFOs, CDC FIFOs and pseudo-CDC FIFOs.

Table 6.7: Composition of the four designs.

	Clocks	CG Actors	Fast-clk Actors	Slow-clk Actors	Synch FIFOs	CDC FIFOs	Pseudo-CDC FIFOs
DNN	1	0	7	0	10	0	0
DNN_a	2	0	4	3	6	4	0
DNN_cg	1	4	7	0	10	0	0
DNN_acg	2	4	4	3	6	4	0
DNN_facg	1	4	7	0	0	0	10

Table 6.8 presents the resource utilization data gathered by the post place&route reports generated by the Xilinx Vivado tool using the aforementioned Zynq board. The asynchronous designs *DNN_a* and *DNN_acg* have the highest overhead in terms of *LUTs* and *REGs*, due to the overhead in the CDC FIFOs. The number of *BUFGs* varies significantly among the five designs, due to two key differences: compared with the baseline DNN, DNN_a needs one additional *BUFG* to control the clock signal of the second clock domain, and DNN_cg and DNN_facg need four additional *BUFGs*, one for each clock-gated actor. DNN_acg needs five additional *BUFGs*, one for each clock domain, and one for each clock-gated actor. The *DNN_auto* design present a negligible overhead.

Table 6.8: Resource utilization. The numbers in parentheses give the percentage of utilization with respect to the resources available on the board.

	LUTs (%)	REGs (%)	BUFGs (%)	BRAMs (%)	DSPs (%)
Available	218600	437200	32	545	900
DNN	9764 (4.47)	6121 (1.40)	1 (3.13)	22 (4.04)	53 (5.89)
DNN_a	9983 (4.56)	6338 (1.45)	2 (6.25)	22 (4.04)	53 (5.89)
DNN_cg	9776 (4.47)	6149 (1.41)	5 (15.63)	22 (4.04)	53 (5.89)
DNN_acg	10086 (4.61)	6357 (1.45)	6 (18.75)	22 (4.04)	53 (5.89)
DNN_facg	9795 (4.48)	6149 (1.41)	5 (15.63)	22 (4.04)	53 (5.89)
DNN_auto	9808 (4.49)	6121 (1.40)	1 (3.13)	22 (4.04)	53 (5.89)

For each of the implemented designs, the switching activity files are generated by post-implementation simulation, and then back-annotated by Vivado to extract the power consumption data. This extracted power consumption data is summarized in Table 6.9.

The clock frequencies of the synchronous designs and that of *Clock Domain 1* in the asynchronous designs are uniformly set to 100 MHz, which is the maximum possible frequency. Concerning *Clock*

Domain 2 of the asynchronous designs, the frequency is set to be 5 MHz, which is 1/20 of the clock frequency in the faster clock domains, according to the profiling results. For the sake of completeness results include also the *DNN_acg* when the frequencies of both clock domains are set to 100 MHz. The frequency of *Clock Domain 2* is specified in Table 6.9 with the suffix "*_num*", where *num* represents the frequency value in MHz.

Table 6.9: Dynamic power consumption. Δ% gives the difference in power consumption compared to the baseline DNN design.

	Dynamic Power Consumption [*W*]				
	CLOCKs	SLICEs	BRAMs	DSPs	Total (Δ%)
DNN	0.039	0.081	0.031	0.028	0.271
DNN_a_5	0.027	0.062	0.025	0.024	0.212 (-21.77)
DNN_cg	0.027	0.063	0.025	0.024	0.214 (-21.03)
DNN_acg_100	0.029	0.063	0.024	0.024	0.215 (-20.66)
DNN_acg_5	0.027	0.062	0.025	0.024	0.209 (-22.88)
DNN_facg	0.026	0.063	0.024	0.024	0.212 (-21.77)
DNN_auto	0.036	0.079	0.031	0.028	0.265 (-2.21)

According to Table 6.9, DNN_a_5 and DNN_facg have equal capabilities in saving power, reducing the total dynamic power consumption by 21.77%, while DNN_cg provides a slightly lower power savings of 21.03%. DNN_acg_5 is the most power-efficient design, with a power savings of 22.88%, while DNN_acg_100 is the least advantageous (among the compared designs), with a power savings of 20.66%.

Through the comparison among DNN_cg, DNN_acg_100 and DNN_facg, which all employ clock domains with frequency 100MHz, it is possible to conclude that DNN_acg_100 saves less power than DNN_cg since the former employs one more *BUFGs*, and also the power saved by switching off the unused parts of the CDC FIFOs does not counterbalance the power overhead due to the additional logic. Furthermore, although DNN_facg involves higher resource utilization than DNN_cg, DNN_facg switches off the idle part of all pseudo-CDC FIFOs, and thus, is actually advantageous compared to DNN_cg.

Through the comparison between DNN_acg_5 and DNN_a_5, both of which employ two clock domains with frequencies of 100 MHz (*Clock Domain 1*) and 5 MHz (*Clock Domain 2*), it comes out that the former design with clock gating saves more power compared to the latter. This is because even though the actors in *Clock Domain 2* are active for a relatively large portion of the time, they can still be switched off by clock gating to save power. Furthermore, according to Table 6.5, the deinterleave actor, which belongs to *Clock Domain 1*, can be switched off for almost 90% of the total execution time. *DNN_auto* is the less power saving design, being able to save only 2.21% of total dynamic power.

Table 6.10 summarizes the execution time, power, and energy consumption of the five designs. All of the clock gated designs present only a 0.003% increase of the execution time, while the asynchronous designs with slow frequency being 5 MHz have an increase of less than 4%. With reference to Figure 6.13, the synchronization circuits of FIFO_r_out and FIFO_g_out introduce a delay in enabling S_1. S_1's Z_{ec} and S_2's Z_{ic} are affected by the slower clock frequency, introducing a delay in enabling the S_2 and *M*&*R*, respectively. After this initial delay, *M*&*R*'s firing rate is close to the original rate, allowing its computation to complete once every 48000ns compared to the original rate of one completion every 48080ns. *DNN_auto* is the only low power implementation that does not increase the execution time, but due to its poor power saving, is not able to save a significant quantity of energy, only 2.21%.

Table 6.10: Execution time, power, and energy. Δ%s give the difference in total graph execution time and energy consumption compared to the baseline DNN design, respectively.

	Execution Time [ns] (Δ%)	Power [W]	Energy [μJ] (Δ%)
DNN	2329165	0.271	631.20
DNN_a_5	2407300 (+3.355)	0.212	510.35 (-19.15)
DNN_cg	2329245 (+0.003)	0.214	498.46 (-21.03)
DNN_acg_100	2329245 (+0.003)	0.215	500.79 (-20.66)
DNN_acg_5	2408100 (+3.389)	0.209	503.29 (-20.26)
DNN_facg	2329245 (+0.003)	0.212	493.80 (-21.76)
DNN_auto	2329165 (+0.000)	0.265	617.23 (-2.21)

Due to the variations in execution time among the different designs, energy consumption is a better metric than power consumption for assessing the efficiency of the designs. In particular, DNN_acg_5 still saves more energy than DNN_a_5 (20.26% versus 19.15%), but it is not the most efficient design in terms of energy consumption. Instead, DNN_cg, DNN_facg and DNN_acg_100, whose execution times are increased of only 0.003%, have lower energy consumption.

These comparisons demonstrate complex relationships among costs and benefits associated with different low power design techniques. The lightweight and modular orientation of the LIDE-V framework allows designers to experiment efficiently and systematically with these relationships. Starting from the same LWDF-V model, it is possible to implement different alternative designs by integrating different DEM implementations or applying the clock gating strategy, without modifying the AIM and AEM implementations. By carrying out experimentation in this way using the model-based framework of LIDE-V, designers can gain quantitative insight into low power signal processing trade-offs in the context of their specific applications and target platforms.

6.5 Chapter Remarks

This Chapter presented a compact set of retargetable APIs, characterized by a standard interface, for lightweight dataflow (LWDF)-based design and implementation using hardware description languages (HDLs). The presented approach emphasizes the natural integration of power management within the proposed APIs. Furthermore, this Chapter also presented LIDE-V, which is an extension of the lightweight dataflow environment (LIDE) that provides support for Verilog-based implementation of the LWDF APIs, along with associated libraries of dataflow actor and edge implementations. LIDE-V facilitates design of and experimentation with alternative implementations of a given LWDF-V model to reveal important insights into system-level trade-offs, and perform multidimensional design optimization.

Analyzing the execution times of LIDE-V actors and their waiting times between firing completion and starting of the next firing, it is possible to systematically identify actors that can benefit the most from clock gating, as well as actors that can operate more efficiently with different clock frequencies. The design techniques introduced in this paper have been demonstrated through an FPGA-based accelerator for a deep neural network (DNN) that performs classification among different types of vehicles. Even if it has been assessed targeting an FPGA, the methodology is target independent and it could be applied also on ASIC technology. Interesting directions for future work include the extension of MDC to automate LWDF-V models and methods introduced in this Chapter.

List of Publications Related to the Chapter

Journal papers

- Tiziana Fanni, Lin Li, Timo Viitanen et al., *Hardware design methodology using lightweight dataflow and its integration with low power techniques.* Journal of Systems Architecture, Volume 78, 2017, Pages 15-29, ISSN 1383-7621.
  DOI: https://doi.org/10.1016/j.sysarc.2017.06.003.

- Lin Li, Carlo Sau, Tiziana FANNI, Jingui Li, Timo Viitanen, Francois Christophec, Francesca Palumbo, Luigi Raffo, Heikki Huttunen, Jarmo Takala, Shuvra S. Bhattacharyya, *An Integrated Hardware/Software Design Methodology for Signal Processing Systems.* Journal of Systems Architecture (2018). DOI: https://doi.org/10.1016/j.sysarc.2018.12.010

Conference papers

- Lin Li, Tiziana FANNI, Timo Viitanen et al., *Low power design methodology for signal processing systems using lightweight dataflow techniques.* 2016 Conference on Design and Architectures for Signal and Image Processing (DASIP), Rennes, 2016, pp. 82-89.
  DOI: 10.1109/DASIP.2016.7853801

Chapter 7

Multi-Grain Reconfiguration on FPGA - A New Level of Flexibility

Cyber-Physical Systems (CPS) operate in increasingly complex and demanding application scenarios, while requiring also high adaptivity levels to satisfy several requirements that usually change over time. The H2020 CERBERO European Project[1] [69] aims at developing a continuous design environment for CPS, including modelling, deployment and verification (http://www.cerbero-h2020.eu/). The efficient support for runtime reconfiguration, taking into account an uncertain environment with changing requirements, is among the CERBERO expected outcomes.

Reconfigurable hardware architectures present high performance and flexibility, being an appealing solution to provide runtime adaptivity support necessary for CPS. The main two possible reconfigurable approaches for runtime adaptivity in FPGA systems are the Dynamic Partial Reconfiguration (DPR) and the Coarse-Grain Reconfiguration (CGR) (see Chapter 2.1). DPR allows to dynamically reconfigure part of an FPGA while the remaining logic (the static part) continues the execution. In the CGR approach all the resources necessary to compute different functionalities are present at the same time on the same datapath, and reconfiguration is enabled by multiplexing them in time. These two approaches present different tradeoff between reconfiguration costs and flexibility. On one hand the DPR architecture offer high flexibility, being able to completely change functionality, paying a cost in terms of time and power consumption proportional to the size of reconfigurable partitions[65]. On the other hand, CGR approach offers fast reconfiguration, requiring only to write some configuration registers, but with limited flexibility (only the functionalities considered at design time are available). The combination of these two hardware reconfiguration approaches brings together the best of both, offering the possibility of achieving different tradeoffs between performance, flexibility and energy consumption, able to offer energy saving in cases were the clock gating or asynchronous and GALS-oriented design methodologies presented in Chapter 6 cannot be applied.

In CERBERO, two tools offer support for hardware reconfiguration. The ARTICo^3 framework provides adaptive and scalable hardware acceleration by exploiting a DPR-based multi-accelerators scheme, based on reconfigurable slots [93]. The MDC design suite, on the other hand, delivers CGR systems, and has been already proved to be a viable solution to enable adaptivity in CPS [97]. The integration of ARTICo^3 and MDC combines together the benefits from both DPR and CGR, leading to the implementation of flexible systems that can adapt to the changing requirements of most CPS

[1]The Cross-layer modEl-based fRamework for multi-oBjective dEsign of Reconfigurable systems in unceRtain hybRid envirOnments (CERBERO) is funded from the European Commission's H2020 Programme under grant agreement *No 732105*

scenarios. Figure 7.1 illustrates an example with four ARTICo^3 slots filled in with MDC CGR accelerators. If the battery availability decreases, it is possible to tune the working points within the slots by reducing precision of CGR accelerators without the need of changing bitstreams. When the battery level is not sufficient to keep the same performance, it is possible to turn off one or more ARTICo^3, to save battery by reducing for example the throughput.

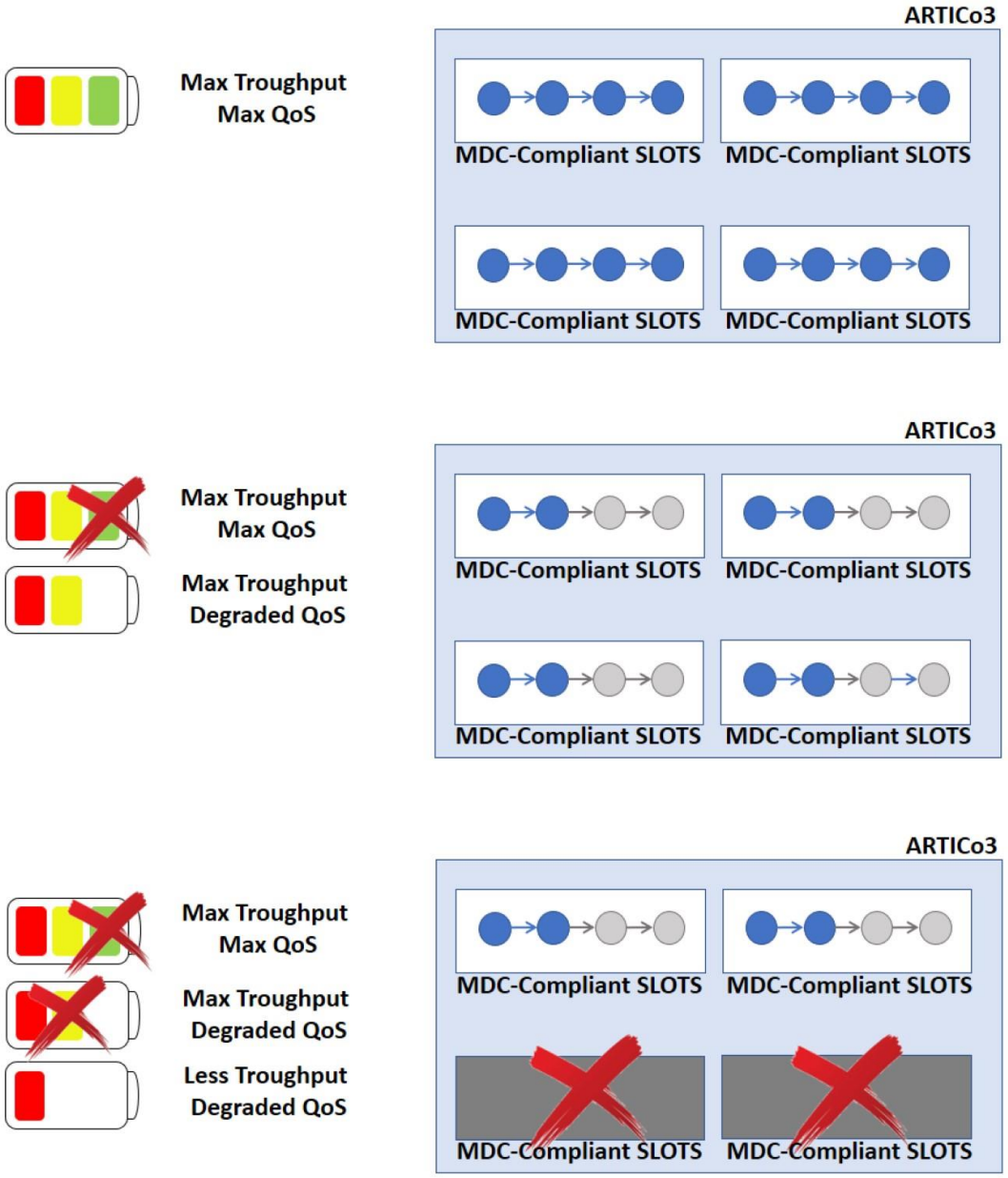


Figure 7.1: Multi-Grain Reconfiguration - The best of DPR and CGR

This Chapter presents the integration of MDC and ARTICo^3 into an automated framework for the development and runtime management of multi-grain reconfigurable hardware systems. The toolchain goes from high-level dataflow specifications to the FPGA implementation of multi-grain adaptive systems, where different reconfigurable partitions of an FPGA are filled in with heterogeneous and irregular application-specific CGR datapaths. The work presented in this Chapter has

been conducted during a six-month visiting period at the Universidad Politécnica de Madrid in a collaboration among the Microelectronics and Bioengineering Lab (EOLAB) (University of Cagliari), the Intelligent system DEsign and Applications (IDEA) Lab (University of Sassari) and the Centro de Electrónica Industrial (Universidad Politécnica de Madrid), into the context of the CERBERO Project.

7.1 SOA on Multi-Grain Reconfiguration

Adaptivity and reconfiguration have been actively studied at different levels of granularity. Network-on-Chip (NoC) technology has been widely exploited in multi-cores systems, to adapt the data traffic to optimize computation and the workload. Several works proposed NoC-based architectures [61, 121, 95, 131]. Li et al. [61] proposed RWiNoC, a reconfigurable wireless-NoC enabled many-core platform to efficiently handle the dynamic workload of the microbial community simulation. The RWiNoC deploys a set of network controllers and wireless interfaces to efficiently adapt to the instantaneous traffic hotspots exhibited by the simulated environment. Xiao et al. [121] exploited an architecture-independent profiling to represent the application as a dynamic application dependency graph (DADG). They proposed a parallelization framework to optimize the scheduling and mapping of the application to different cores in an NoC-based multi-core system for parallel execution, minimizing the inter-core traffic overhead according to the considered DADG. The research of Salvador et al. [95] leverages on the Software Defined Network on Chip (SDNoC) approach, where the control network and the data network are physically separated. Salvador et al. introduced a Software Defined Network Controller (SDNC) capable of reconfiguring the infrastructure at the data forwarding and data processing levels of a SDNoC, allowing the execution of different algorithms at runtime. Zhang et al. [131] proposed a a two-stage variation-aware task mapping scheme for multi-core NoCs with redundant cores. At design time, the proposed algorithm is able to generate a set of candidate task mapping solutions, which considers the process variations and covers the timing requirements for different chips. Then at the runtime, the task scheduler would select a most effective one according to the actual situation of the chip. All of the above described works exploit the reconfiguration offered by NoC technology to optimally distribute data workload among multiple cores, according to traffic data or cores workload, and none of them include hardware acceleration, which is usually required when execution efficiency of multicores is not enough (e.g. in terms of time, power or area). With respect to those works, the multi-grain reconfiguration presented in this chapter is meant to provide a multi-accelerators architecture, able to offer different tradeoff according to requirements (e.g, performance versus energy consumption versus fault-tolerance). Reconfiguration is meant not only inside the accelerator, to reconfigure the functionality (e.g., switching among different algorithms) or working point of the algorithm (e.g., switching among different power profiles), but also at a coarser level, to change the number of accelerators offering a flexible scalability for performance or fault-tolerance purpose.

Different works presented multi-grain reconfigurable architectures with homogeneous structure (e.g., [1, 109, 3, 33]). Amagasaki et al. [1] focused their studies on the development of a Variable Grain Logic Cell (VGLC) architecture that adjusts computational granularity according to the application. The basic logic element (BLE) of a VGLC is a hybrid cell (HC) that can be configured to work as either a two input 1-bit full adder or a 2-input LUT according to the computation (the full adder and the LUT share some common logic). Together with the HC, the BLE includes MUXes and EXORs. Using a VGLC with four BLEs, the VGLC can implement both nibble bit ALU and 4-input random logic, having five different operation modes. The HoneyComb architecture [3] is an adaptable dynamically reconfigurable cell array. Cells are composed of a routing unit and a functional module. Routing units, responsible for connecting neighbours, compose the reconfigurable interconnection network. The specification of every component within the array can be enabled, disabled, or modified using dynamic partial reconfiguration. Diniz et al. [33] proposed runtime accelerator binding for

tile-based architectures, adopting multi-grain reconfiguration within the tiles of tile-based processor. Each tile consists of multiple CGR and FGR elements. The number of reconfigurable elements inside each tile and in the whole architecture is a design time decision and the overall structure of the tile-based system is fixed to a mesh-based one. Given an architectural configuration, a communication-minimizing binding for datapaths of custom instructions is determined at runtime, employing datapath reusing and inter-tile communication cost estimation. All the above-mentioned approaches involve limited CGR arrays, where the processing elements are in most of the cases identical and not directly derived from the applications to be accelerated. This lack of specialization may limit performance, which is made even worse if fixed interconnection infrastructures are used. The design approach followed in this Chapter is application-to-hardware, offering a multi-grain architecture were the CGR accelerators involved are application-based.

Some works focused on multi-grain architectures with heterogeneous structure [118, 109, 129, 64]. Morpheus [118] is a heterogeneous reconfigurable system on chip that integrates application-oriented reconfigurable cores for implementing applications belonging to different domains on the same hardware architecture, and a toolchain that eases the applications implementation through a software oriented approach. MORPHEUS is built around three reconfigurable cores of different granularity: (1) a data processing architecture based on a hierarchical array of CGR 16-bit computing elements communicating through a matrix of configurable data channels; (2) a reconfigurable processor composed by a RISC processor coupled with a mid-grain reconfigurable datapath, that exploits instruction level parallelism for a wide range of applications; (3) an embedded FPGA, suitable for FGR algorithm or arbitrary logic implementation. The interconnect system is a combination of a Network-on-Chip and a conventional bus, able of supporting different interconnect streams depending on each application requirements. The DeSyRe SoC [109] leverages on a multi-grain texture containing different sub-components surrounded by reconfigurable interconnects. The DeSyRe framework relies on a flexible and dynamically reconfigurable hardware substrate to isolate, replace and (when possible) correct design and manufacturing defects as well as other permanent faults due to aging. It involves either CGR or FGR units that can be replaced when defective of for functional purpose. In the CGR case the substitutable unit can be an entire sub-component (e.g. a microprocessors pipeline stage), while in the latter case an FPGA logic cell. In particular CGR logic is not very defect tolerant but its efficiency in terms of resources and power is excellent; on the contrary, FGR logic can be strongly defect tolerant but it has relevant performance, power and cost overheads. The possibility of implementing functionalities on both CGR and FGR logic enables the quick adaptation of the system to different faulty situations or to better meet design constraints and application requirements, such as throughput or load balancing. Repair-oriented adaptivity also provided with re-routing, re-targeting functionalities on unused sub-component. Yuan at al. [129] presented a multi-grain FPGA aimed for mobile computing and focused on two key steps towards higher efficiency: interconnection network and CGR digital signal processors. The chip incorporates FGR logic blocks, medium-grain digital signal processors along with reconfigurable block RAMs, and two CGR kernels. Liu et al. [64] proposed a hybrid-grained reconfigurable architecture (HReA) to process 13-Dwarfs computation [8] (a dwarf is an algorithmic method that captures a pattern of computation and communication). HReA combines a 32-bit CGR datapath with a 1-bit FGR datapath to accommodate co-existence of multiple computing granularities in 13-Dwarfs. The two datapaths with different granularities can interact with each other in an arithmetic logic unit. Architectures described above offer higher flexibility than homogeneous architecture but, as for homogeneous architecture, none of them follow an application-to-hardware design for CGR kernels. Moreover, to the best of our knowledge, they do not offer instruments to partition functionalities between FGR and CGR substrates.

This Chapter proposes a multi-grain architecture flexible enough to be suitable to support different adaptivity types: functional, non-functional and even repair-oriented ones. Moreover, with respect to the above-mentioned works in literature, it is not simply meant to present a novel architecture, rather it aims at building proper hardware abstractions and at designing an infrastructure for

the design of the different parts of the system, for their deployment and runtime management.

7.2 Methodology - Multi-Grain Adaptivity

As explained in Chapter 2.1, the FGR approach is highly flexible, offering the possibility of completely change the system behaviour. This flexibility is not for free, changing the bitstream of the system requires high time and power consumption. The Dynamic Partial Reconfiguration (DPR) mitigates those limitations, having the ability of modifying blocks of logic by downloading partial bitstreams while the remaining logic continues the execution. With DPR architectures the deployed circuit is always optimized in terms of resource usage and frequency, meaning that only the logic effectively required for the current computation is instantiated, but reconfiguration still requires a certain amount of time and an associated power consumption that is proportional to the size of reconfigurable partitions[65]. In CGR computing cores the reconfiguration is virtual. They involve a fixed set of functionalities and the resources belonging to all the possible configurations are always instantiated in the substrate. Reconfiguration is guaranteed by the insertion of multiplexers that allow the core to switch among functionalities. This brings high-performance and fast reconfiguration, at the cost of a degradation in terms of maximum operating frequency, area utilization, and flexibility (being able to compute only the functionalities the system has been designed for). The combination of these two hardware reconfiguration approaches delivers the best of both and better cope with the challenging CPS adaptivity needs and face the changing requirements typical of such kind of systems.

The ARTICo^3 framework [93] provides adaptive and scalable hardware acceleration, actively altering the computing substrate to change the available functionality using DPR. It can be exploited to achieve user-driven runtime tradeoffs between performance, energy efficiency and fault tolerance by means of accelerator replication. However, this flexibility is obtained paying the cost of DPR reconfiguration, that requires long time and high energy consumption. When frequent changes among functionalities are required, this cost may easily become no longer affordable. On the other hand, the MDC tool [97] delivers automatic generation and management of CGR computing cores based on the dataflow model of computation. MDC accelerators offer high-speed low-energy reconfiguration, enabled by simply writing some configuration registers. However the flexibility is limited to the available configurations (decided at design time) and the only way to provide parallelism is instantiating multiple accelerators, thus paying the cost of the extra logic that is always present.

A combination of ARTICo^3 and MDC flows delivers the best of both CGR and DPR approaches, enabling adaptive multi-grain reconfigurable fabrics, which can meet the changing functional and non-functional requirements of CPS designs. Figure 7.2 shows an overview of the MDC-ARTICo^3 integrated toolchain: The hardware generation flow starts from high-level dataflow descriptions of the configurations/behaviours to be implemented in the configurable logic, and the integrated toolchain derives the corresponding CGR HDL computational kernel, properly wrapping it with the glue logic necessary to serve as an ARTICo^3 DPR reconfigurable partition. Both reconfiguration mechanisms are transparently managed by the user code running in a host processor. With respect to the standalone MDC and ARTICo^3 flows, an adaptation step (*Kernel Adapter*) is needed to make the MDC-generated kernels compliant with kernels expected by ARTICo^3 *Wrapper Automation* step. The rest of this Section gives a deeper description of ARTICo^3 framework (Section 7.2.1); discusses the modifications necessary in MDC 7.2.2; and present the *Kernel Adapter* (Section 7.2.3).

7.2.1 The ARTICo^3 Framework

The use of SRAM-based FPGAs has merged the best of two worlds (i.e. hardware and software), enabling systems with software-like flexibility while keeping high-performance benefits of dedicated

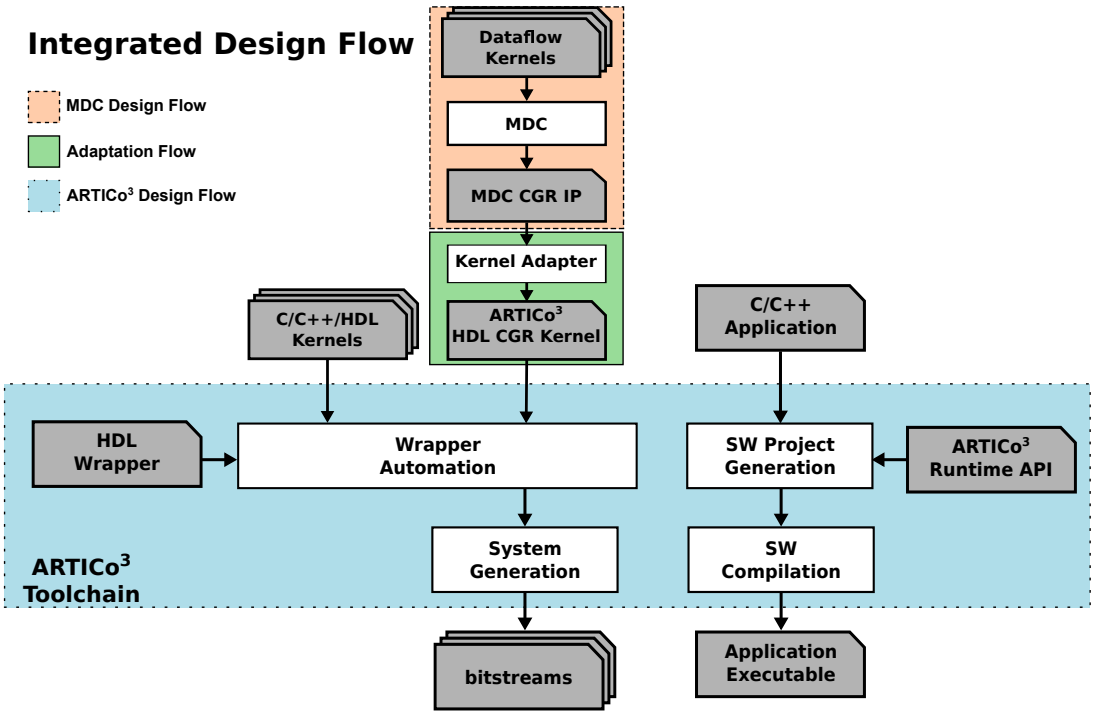


Figure 7.2: Integrated Hardware Design Flow

hardware-based processing. The specific technology that supports this feature is the DPR. The ARTICo^3 framework is composed of three main components:

1. a hardware architecture that exploits a DPR-enabled multi-accelerator computing scheme, in which accelerators can occupy one or more slots;

2. a toolchain to automatically generate the DPR-enabled system starting from the user-defined hardware accelerators and software applications;

3. a *Runtime Library* to transparently manage application execution and computation offloading to the hardware accelerators.

The ARTICo^3 architecture exploits DPR in high-performance embedded systems that use a processor-coprocessor approach [93]. However, instead of relying only on one application-specific hardware accelerator for each task, as it has been traditionally done, the computing fabric supports a multi-accelerator based computing scheme. Similarly to embedded GPUs supporting general purpose computing, the ARTICo^3 computing fabric can operate in SIMD-like fashion (Single Instruction Multiple Data), where different copies of a given hardware accelerator work on different sets of input data. In addition, module replication using DPR can be also used in combination with a configurable datapath to increase fault tolerance in the reconfigurable partitions, using two or even three copies of an accelerator performing the same computation over the same input data where the results are retrieved through a voting unit to mask possible errors. It is important to highlight that ARTICo^3-based computing requires previous hardware/software partitioning in order to identify computing-intensive data-parallel tasks to be implemented as hardware accelerators.

ARTICo^3-based hardware accelerators are connected to the communication infrastructure in the system using a custom gateway, called Shuffler, which is able to dynamically alter its internal datapath to meet specific requirements of computing performance or energy consumption and fault tolerance. The gateway hides custom point-to-point interfaces (reconfigurable partition) behind a

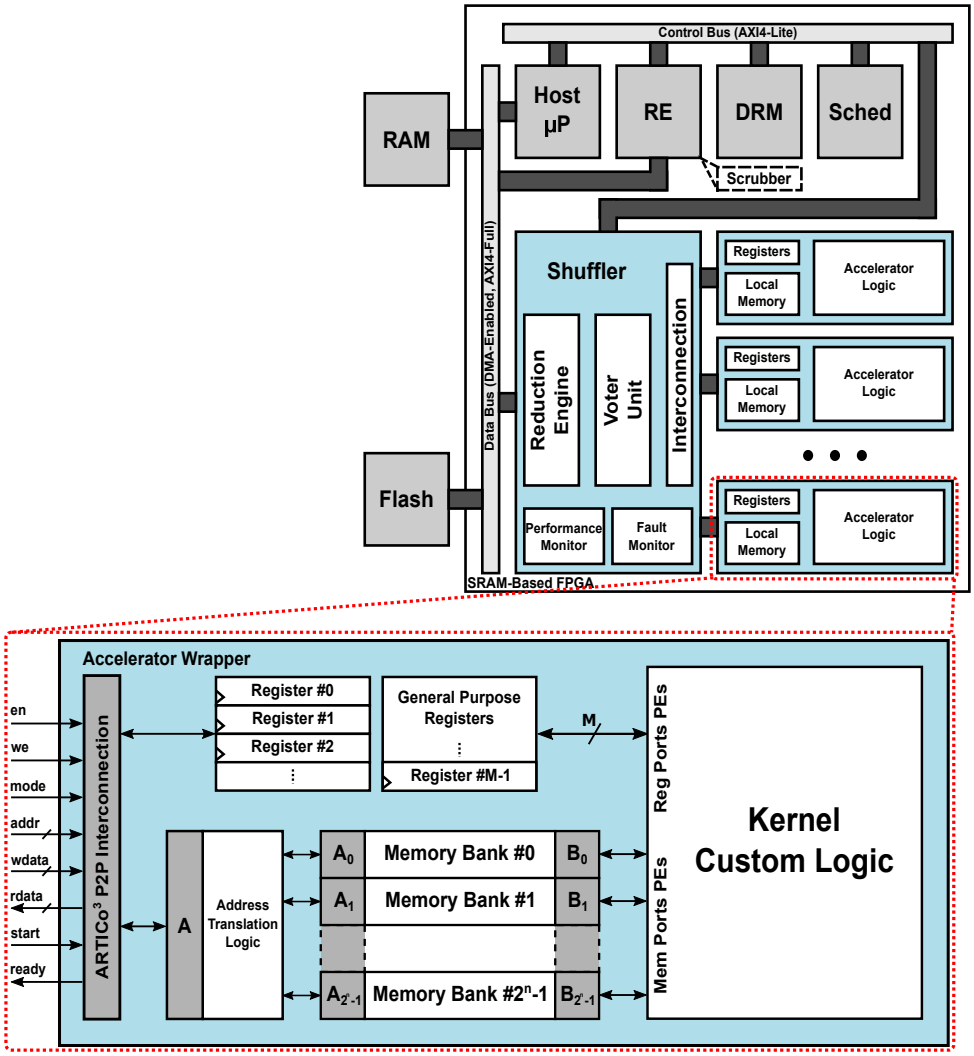


Figure 7.3: Schematic view of the ARTICo^3 Architecture, with a zoom on the ARTICo^3 Wrapper (as presented in Rodríguez et al. [93]).

standard AXI4 interface [123] (static partition). Plug-and-play capabilities are enabled in user-defined custom accelerators by instantiating them in wrapper modules embedding a local memory (divided in banks) and a register bank (for configuration purposes). Figure 7.3 depicts a schematic view of the ARTICo^3 architecture with a zoom on the ARTICo^3 slot wrapper.

The automated toolchain eases the design of ARTICo^3-based system, making it accessible to embedded system designer with little knowledge of DPR-based design. Figure 7.4 gives a schematic view of the toolchain. On the left side it shows the hardware design flow, which encapsulates the user-defined application specification into the standard wrapper with the glue logic to communicate with the rest of the system. Generation of hardware accelerators is automatically carried out by ARTICo^3 toolchain that builds the fabric starting from either C/C++ code (running automatically Vivado HLS) or HDL application specifications. Then the toolchain generates the system, embedding the accelerators into the rest of the hardware system and generating the required bitstreams.

On the software design flow, the toolchain generates the required software project starting from the user-defined software application and combining it with the API to access the underlying ARTICo^3 *Runtime Library*; the result is a customized Makefile to transparently build the application executable. The *Runtime Library* manages applications execution and computation offloading to the hardware

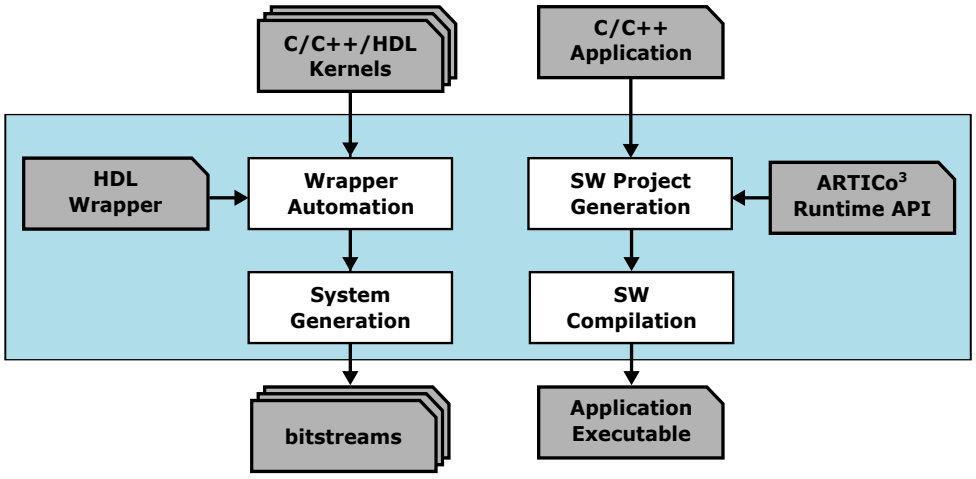


Figure 7.4: ARTICo^3 toolchain.

accelerators. This runtime library, running under Linux and exploiting DMA for data transfers, provides support to establish runtime tradeoffs between computing performance and energy consumption.

7.2.2 New Coprocessor Generator for MDC

The MDC *Coprocessor Generator* presented in Chapter 3, able of automatically providing a Xilinx compliant runtime reconfigurable coprocessor, exploited in its first version the Xilinx ISE Design Suite. Current version ARTICo^3 toolchain works with Xilinx Vivado Design Suite. In order to integrate MDC and ARTICo^3 design flows, it has been necessary, as a first step, to completely modify the MDC *Coprocessor Generator* to exploit Xilinx Vivado Design Suite to generate runtime reconfigurable coprocessors.

Alignment to Xilinx Vivado Design Suite

Figure 7.5 illustrates the new MDC *Coprocessor Generator* flow. MDC generates the multi-dataflow (*Multi-flow IR*) merging the input dataflow specifications as described in Section 3.1 (1), then starting from the generated multi-dataflow MDC generates the corresponding CGR core (2). In parallel, starting from the composed multi-dataflow, MDC generates the files and the necessary logic to embed the computing core into a configurable Template Interface Layer (TIL) (3). Finally, to easy deploy and use the coprocessor, MDC provides the Xilinx Vivado scripts to automatically embed the logic into a processor-coprocessor architecture and the software drivers to ease its use (4). Following Sections give details on the TILs composition that, as in the *Coprocessor Generator* presented in Chapter 3, can be for memory-mapped or stream coupling, on the software drivers generated accordingly, and on the generated scripts for Xilinx Vivado.

Template Interface Layer: with respect to the TILs described in Chapter 3, the TILs generated by the new *Coprocessor Generator* require a totally different interface, since they exploit the AXI communication protocol [123]. Figure 7.6 shows the architecture of the memory-mapped TIL (mm-TIL) whose main blocks are: the configuration registers bank, and one local memory, one front-end and one back-end for each I/O port. The local memory and the configuration registers bank work as described in Chapter 3.4. However, in this new TIL the configuration registers are written through the AXI4-Lite (AXI_lite in Figure 7.6) interface which is generally used for simple, low-throughput

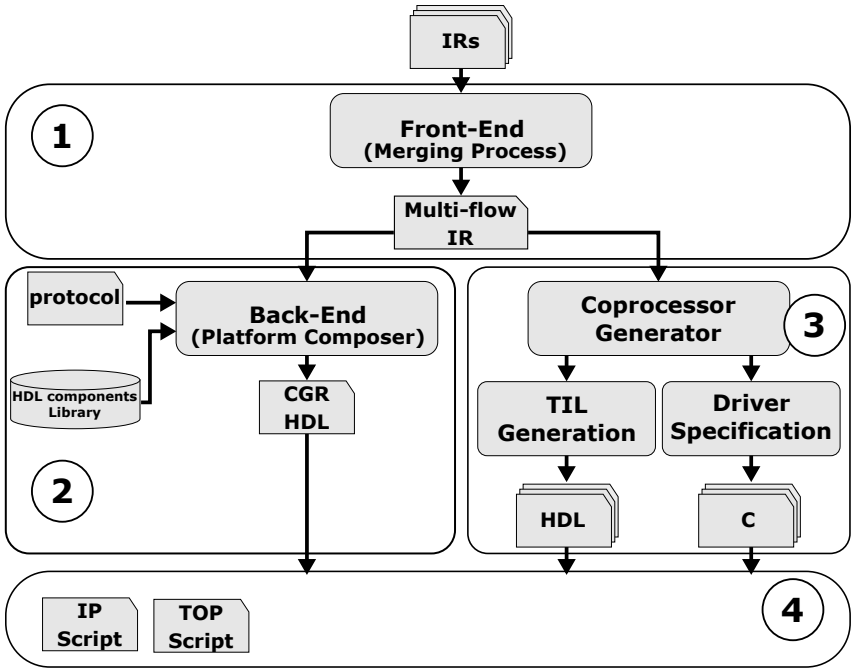


Figure 7.5: Design flow overview.

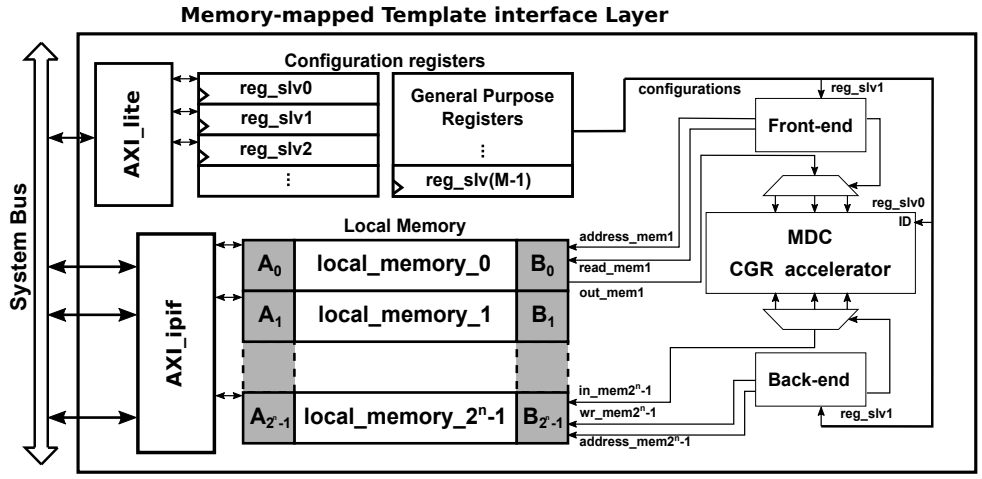


Figure 7.6: Architecture of the memory-mapped Template Interface Layer (mm-TIL).

memory-mapped communication, while memory banks are written through the AXI4-full (AXI_ipif in Figure 7.6) generally used for high-performance memory-mapped requirements.

Figure 7.7 depicts the stream-based TIL (s-TIL) architecture. With respect to Chapter 3, this TIL has been modified to leverage on a the AXI4-Stream communication protocol, generally used for high-speed streaming data. The configuration registers bank, as in the mm-TIL, saves the coprocessor configuration. In the s-TIL the front-end and back-end are not present anymore, however it is necessary a module to generate the `last` signal.

Driver Specification: the different organization of data, made necessary to modify the generation of the software drivers. At high level, they offer an interface that masks the system configuration complexity, providing a C function for each configuration of the CGR coprocessor. Listing 7.1 shows the interface driver for both memory-mapped and stream-based coprocessor drivers, for one configura-

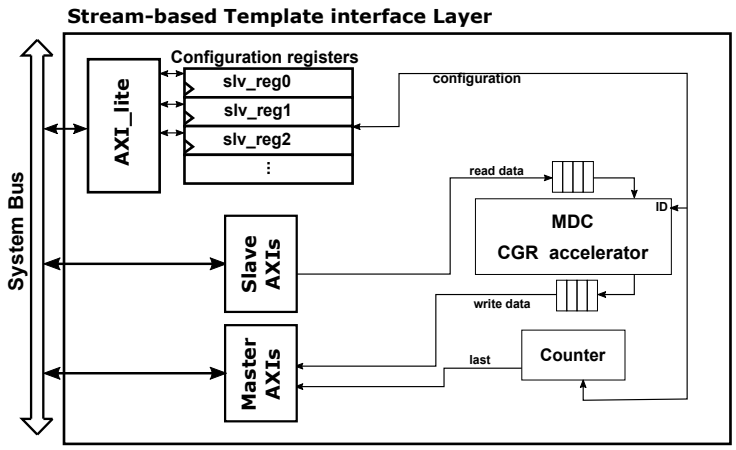


Figure 7.7: Architecture of the stream-based Template Interface Layer (s-TIL).

tion of a CGR coprocessor.

`data_<port_name>` and `size_<port_name>` are respectively input (or output) port and the number of data related to that port, in the considered example there are three ports: `in_size`, `in_pel` and `out_pel`.

Listing 7.1: Coprocessor drivers interface.

```
//////////////////////////////////
//Memory-Mapped Interface Driver
int mm_accelerator_roberts(
// port out_pel
int size_out_pel, int* data_out_pel,
// port in_pel
int size_in_pel, int* data_in_pel,
// port in_size
int size_in_size, int* data_in_size
)
...

//////////////////////////////////
// Stream-Based Interface Driver
int s_accelerator_roberts(
// port out_pel
int size_out_pel, int* data_out_pel,
// port in_pel
int size_in_pel, int* data_in_pel,
// port in_size
int size_in_size, int* data_in_size
)
...
```

It is clear as the interfaces for the two cases, memory-mapped and stream, are identical. This allows software designer with little knowledge of hardware design to easily use the generated processor-coprocessor systems, without considering the underlying processor-coprocessor coupling. Then the body of the function manages communication between the host processor and the coprocessor (see Listing 7.2). For each I/O port of the reconfigurable computing core, a configuration word is written into the proper configuration register (`*(config + 1) = size_<port_name>`). Then, the indicated amount of data (`size_<port_name>`) for each input port involved in the current computation is sent to the corresponding local memory or to the input FIFO according to the chosen coupling —

memory-mapped or stream-based (see lines under `//send data port in_size` comment). At last, as the processor can read back the results into the processor from the output ports (see lines under `//receive data port out_pel` comment). In the case of memory-mapped coupling, the processor need to monitor through polling a configuration register where a done data is stored at the end of the computation. In the case of stream coupling a done signal is not necessary, since the processor only needs to evaluate the state of the output FIFOs.

Listing 7.2: Coprocessor drivers body.

```
//////////////////////////////
// Memory-Mapped Body Driver
...
// configure I/O
*(config + 1) = size_in_size;
...
// send data port in_size
*((volatile int*) XPAR_AXI_CDMA_0_BASEADDR + (0x04>>2)) = 0x00000002; // verify idle
*((volatile int*) XPAR_AXI_CDMA_0_BASEADDR + (0x18>>2)) = (int) data_in_size; // src
*((volatile int*) XPAR_AXI_CDMA_0_BASEADDR + (0x20>>2)) =
XPAR_MM_ACCELERATOR_0_MEM_BASEADDR + MM_ACCELERATOR_MEM_1_OFFSET; // dst
*((volatile int*) XPAR_AXI_CDMA_0_BASEADDR + (0x28>>2)) = size_in_size*4; // size [B]
while((*((volatile int*) XPAR_AXI_CDMA_0_BASEADDR + (0x04>>2)) & 0x2) != 0x2);
...

// receive data port out_pel
*((volatile int*) XPAR_AXI_CDMA_0_BASEADDR + (0x04>>2)) = 0x00000002; // verify idle
*((volatile int*) XPAR_AXI_CDMA_0_BASEADDR + (0x18>>2)) =
XPAR_MM_ACCELERATOR_0_MEM_BASEADDR + MM_ACCELERATOR_MEM_3_OFFSET; // src
*((volatile int*) XPAR_AXI_CDMA_0_BASEADDR + (0x20>>2)) = (int) data_out_pel; // dst
*((volatile int*) XPAR_AXI_CDMA_0_BASEADDR + (0x28>>2)) = size_out_pel*4; // size [B]
while((*((volatile int*) XPAR_AXI_CDMA_0_BASEADDR + (0x04>>2)) & 0x2) != 0x2);
...

////////////////////////////
// Stream-Based Body Driver
...
// configure I/O
*((int*) (XPAR_S_ACCELERATOR_0_CFG_BASEADDR + 1*4)) = size_out_pel;

// start execution
*(config) = 0x2000001;

// send data port in_size
*((volatile int*) XPAR_AXI_DMA_0_BASEADDR + (0x00>>2)) = 0x00000001; // start
*((volatile int*) XPAR_AXI_DMA_0_BASEADDR + (0x04>>2)) = 0x00000000; // reset idle
*((volatile int*) XPAR_AXI_DMA_0_BASEADDR + (0x18>>2)) = (int) data_in_size; // src
*((volatile int*) XPAR_AXI_DMA_0_BASEADDR + (0x28>>2)) = size_in_size*4; // size [B]
while(((*((volatile int*) XPAR_AXI_DMA_0_BASEADDR + (0x04>>2))) & 0x2) != 0x2);
...

// receive data port out_pel
*((volatile int*) XPAR_AXI_DMA_0_BASEADDR + (0x30>>2)) = 0x00000001; // start
*((volatile int*) XPAR_AXI_DMA_0_BASEADDR + (0x34>>2)) = 0x00000000; // reset idle
*((volatile int*) XPAR_AXI_DMA_0_BASEADDR + (0x48>>2)) = (int) data_out_pel; // dst
*((volatile int*) XPAR_AXI_DMA_0_BASEADDR + (0x58>>2)) = size_out_pel*4; // size [B]
while(((*((volatile int*) XPAR_AXI_DMA_0_BASEADDR + (0x34>>2))) & 0x2) != 0x2);
```

...

Coprocessor Deployment: In order to integrate and deploy the peripheral as a standard Xilinx IP, MDC provides an automatic script for Xilinx Vivado Design Suite. The inputs for the script are the HDL description of generated TIL, including TIL submodules (config registers, local memories, front end, ...) and the CGR core modules (`add_files $hdl_files_path`), any required HDL library (`set_property library caph`) and the generated drivers (`ipx::add_file_group -type software_driver`); whereas, the output is the resulting Xilinx ready-to-use IP comprehensive of software drivers.

Listing 7.3: IP Generation Script.

```
##########################
# IP Settings
##########################

...
# FPGA device
set partname "xc7z020clg400-1"
set boardpart "digilentinc.com:arty-z7-20:part0:1.0"

# Design name
set ip_name "mm_accelerator"
set design $ip_name

##########################
# Create IP
##########################

create_project -force $design $ipdir -part $partname
set_property board_part $boardpart [current_project]
set_property target_language Verilog [current_project]

add_files $hdl_files_path
import_files -force

set files [glob -tails -directory $ipdir/.../lib/caph/ *]
foreach f $files {
set name $f
set_property library caph [get_files  $ipdir/.../lib/caph/$f]
}

set_property top $ip_name [current_fileset]

ipx::package_project -root_dir $ipdir -vendor user.org\
-library user -taxonomy AXI_Peripheral

ipx::add_address_block s00_axi_reg\
[ipx::get_memory_maps s00_axi -of_objects [ipx::current_core]]
ipx::add_address_block s01_axi_mem\
[ipx::get_memory_maps s01_axi -of_objects [ipx::current_core]]
...

file copy -force $iproot/drivers $ipdir
set drivers_dir drivers
ipx::add_file_group -type software_driver {} [ipx::current_core]
...
```

```
set_property core_revision 3 [ipx::current_core]
ipx::create_xgui_files [ipx::current_core]
ipx::update_checksums [ipx::current_core]
ipx::save_core [ipx::current_core]
set_property  ip_repo_paths $ipdir [current_project]
update_ip_catalog
close_project
```

MDC provides another script to instantiate the generated IP into a processor-coprocessor system, within the Vivado environment. According to the user choice, the host processor can be a hard-core (ARM processor) or a soft-core (Microblaze); in the considered example an ARM processor is instantiated (`create_bd_cell -type ip -vlnv ... processing_system7_0`). The communication between processor and coprocessor can be managed either with or without a Direct Memory Access (DMA) module; in the considered example for a memory-mapped communication through DMA, the AXI Central Direct Memory Access (CDMA) module is instantiated (`create_bd_cell -type ip -vlnv ... axi_cdma_0`).

Listing 7.4: Top Design Generation Script

```
############################
# Settings
############################

...
# FPGA device
set partname "xc7z020clg400-1"
set boardpart "digilentinc.com:arty-z7-20:part0:1.0"

# Design name
set design system
set bd_design "design_1"

############################
# Create Project
############################
create_project -force $design $projdir -part $partname
set_property board_part $boardpart [current_project]
set_property target_language Verilog [current_project]
set_property  ip_repo_paths $ipdir [current_project]
update_ip_catalog -rebuild -scan_changes
############################
#create block design
create_bd_design $bd_design

# Zynq PS
create_bd_cell -type ip\
-vlnv xilinx.com:ip:processing_system7:5.5 processing_system7_0
...

# accelerator IP
create_bd_cell -type ip -vlnv user.org:user:$ip_name:$ip_version $ip_name\_0

apply_bd_automation -rule xilinx.com:bd_rule:axi4\
-config {...}  [get_bd_intf_pins $ip_name\_0/s00_axi]

# CDMA
create_bd_cell -type ip -vlnv xilinx.com:ip:axi_cdma:4.1 axi_cdma_0
```

```
set_property -dict [list CONFIG.C_INCLUDE_SG {0}] [get_bd_cells axi_cdma_0]

apply_bd_automation -rule xilinx.com:bd_rule:axi4\
-config {...}  [get_bd_intf_pins axi_cdma_0/S_AXI_LITE]
...

make_wrapper -files [get_files $projdir/.../design_1.bd] -top
add_files -norecurse $projdir/.../hdl/design_1_wrapper.v
...
```

Modification in MDC for ARTICo^3 Compliant Operation

Given the new MDC *Comprocessor Generator*, only little changes in the mm-TIL have been necessary to make the operation of the generated MDC CGR core compliant to ARTICo^3 slots. Indeed the front-end and back-end modules of the mm-TIL have been modified to consider the same control signal used in ARTICo^3 slots, with the same synchronization. In particular, previous version of the mm-TIL needed an `enable` signal active for the duration of the computation, now it considers a `start` signal that is high only when the computation has to start.

7.2.3 Kernel Adapter

Comparing the ARTICo^3 slot wrapper shown in the bottom part of Figure 7.3 with the MDC mm-TIL shown in Figure 7.6, it is evident as common elements are present, indeed both of them embed banks of memories and registers. In the case of the standard ARTICo^3 kernel (ARTICo^3 slot) the logic is directly interfaced with banks of configuration registers and local memories. In the case of MDC mm-TIL, the multi-functional kernel (MDC CGR accelerator) is interfaced with memories and registers through the front-end and back-end logic. Thus, if the logic generated by MDC is embedded in a kernel module with a ARTICo^3-compliant interface, as expected by ARTICo^3 design flow, it is possible to feed ARTICo^3 slots with MDC CGR accelerators. The adaptation kernel script is depicted in Listing D.1, in Appendix D.

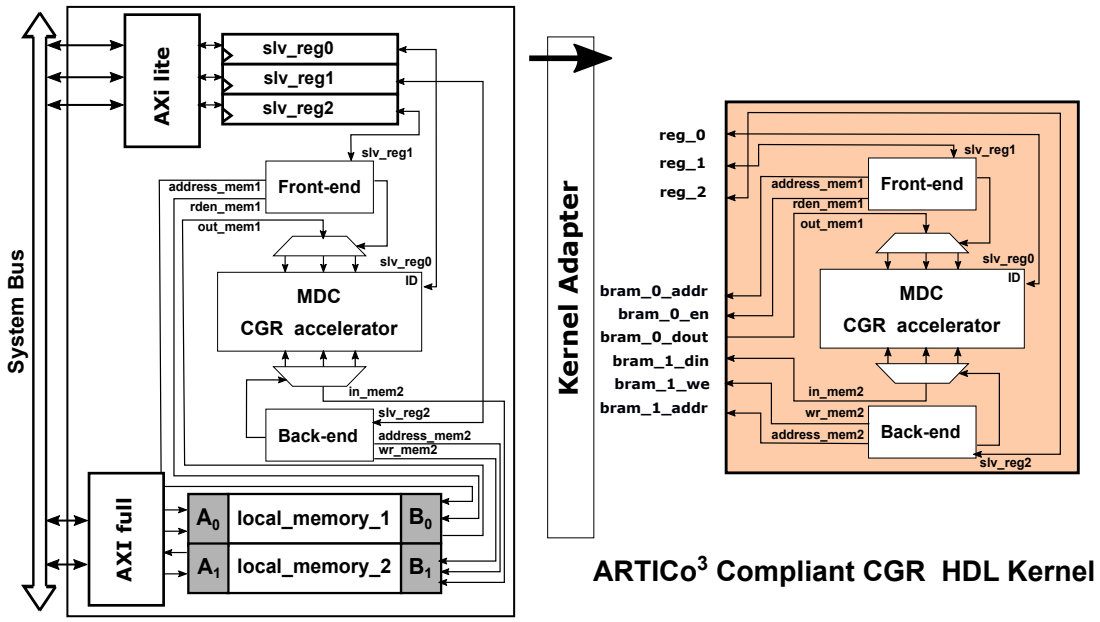


Figure 7.8: *Adaptation Flow* from an MDC- to a ARTICo^3-compliant CGR IP.

Figure 7.8 shows an example where the MDC-generated mm-TIL, with one input port and one output port, requiring three configuration registers and two memory blocks, is modified through the *Adaptation Flow*. The *Adaptation Flow* parses the mm-TIL to remove all interfacing logic (AXI-lite interface, AXI-full interface, local memories and configuration register bank). The remaining logic is then instantiated in an HDL wrapper whose external interface is the one required normally in a standard ARTICo^3 kernel specification. During this step, the *Adaptation Flow* takes also care of properly connecting the ports of the new HDL wrapper to the internal CGR-kernel logic.

7.2.4 Step-by-Step Example

Figure 7.9 illustrates the whole MDC-ARTICo^3 design flow, through a Steb-by-Step example that considers three input dataflow networks. (1) Firstly, MDC merges the user-defined dataflow specifications and generates the CGR computing core as described in Chapter 3.1, that is embedded in the mm-TIL described in Section 7.2.2. (2) Then, the generated mm-TIL is modified by the *Kernel Adapter* which delivers an HDL ARTICo^3-compliant CGR kernel. (3) Finally, the ARTICo^3 framework processes the input HDL CGR kernel to implement the whole reconfigurable processing system (see Section 7.2.1). (4) The bottom part of Figure 7.9 depicts an example of multi-grain reconfiguration. In particular, in this example the CGR approach offered by MDC is exploited for low-power fast-switching of functionality, while the DPR supported by ARTICo^3 is exploited for changing the number of slots working in parallel to increase the throughput.

7.3 Assessment

The multi-grain reconfiguration approach has been evaluated through a proof of concept edge detection application involving two different algorithms: *Sobel* and *Roberts*. This use-case enables the possibility of evaluating the advanced adaptivity features regarding functional (changing the edge detection algorithm) and non-functional (tradeoff in terms of execution time and energy) requirements, acting as a demonstrative example of CPS behaviour. Assessment results have been collected targeting a custom Zynq-7000 board, based on the XC7Z020CLG484-1 device, with integrated power monitoring circuitry.

7.3.1 Test Case: Edge Detection - Sobel and Roberts algorithms

Edge detection algorithms [30] estimate the magnitude and the orientation of edges on a image and are widely used in several application fields such as image segmentation, image compression, computer vision and security. In a digital image, the boundary of an object is a difference of its pixels intensity levels with respect to the surrounding pixels. Popular methods for edge detection are the search-based methods. Detection in search-based methods consists of two steps: (1) computing a measure of edge strength, usually involving discrete first-order differentiation operators such as the gradient magnitude; (2) searching for local directional maxima of the gradient magnitude, usually adopting the gradient direction. To assess the proposed architecture, Sobel and Roberts detectors have been used. These operators are applied to evaluate the gradient image $G = k * A$, given by the convolution of the kernel k with the source image A (k is 3x3 for Sobel and 2x2 for Roberts). The G function, corresponding to the magnitude of the edge, is calculated as

$$G = \sqrt{G_x^2 + G_y^2}$$

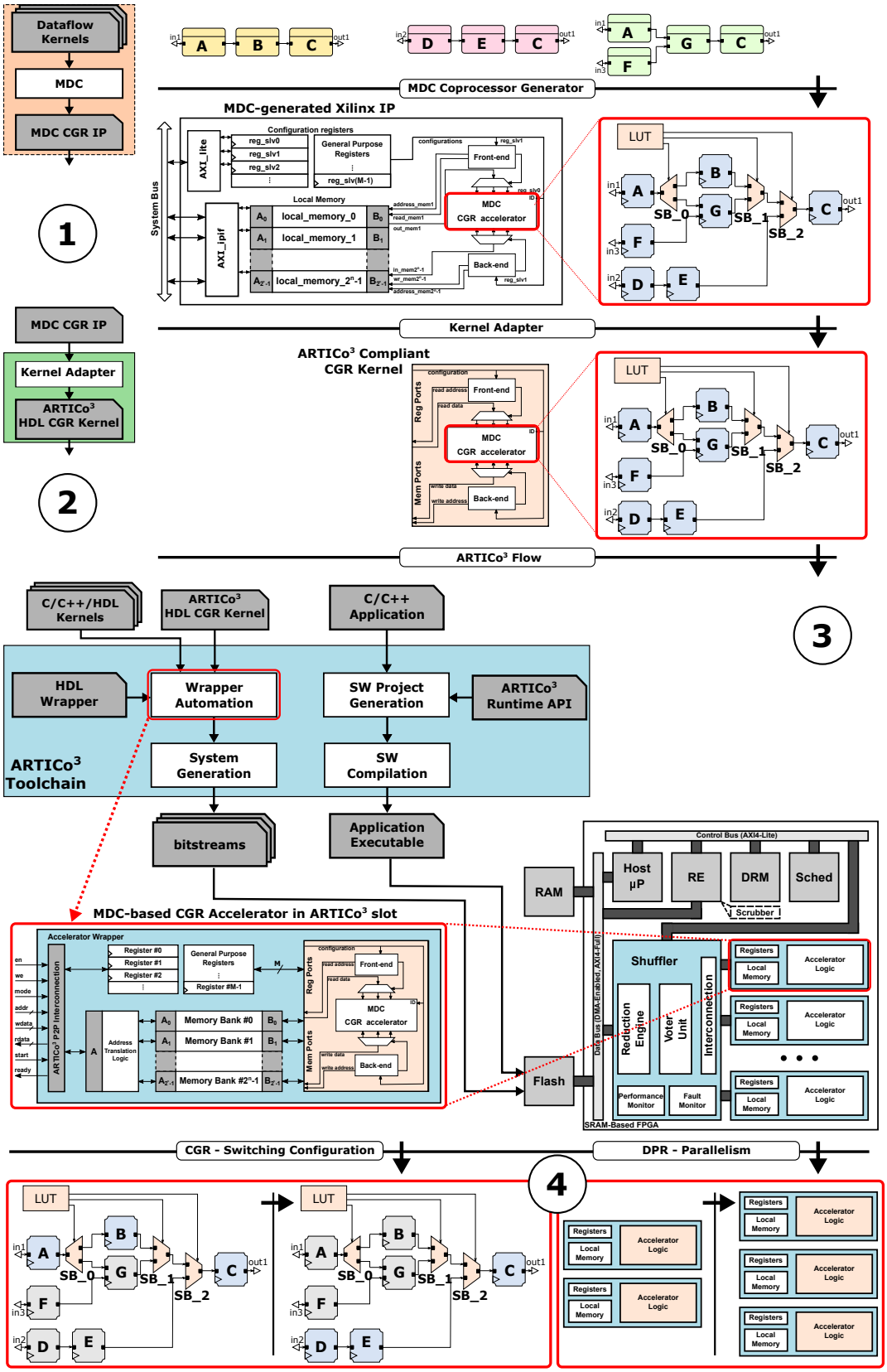


Figure 7.9: Integrated Design Flow - Step-by-step Example

where G_x and G_y are obtained as:

$$G_x = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} * A \quad , \quad G_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * A$$

for the Sobel case [133], and as:

$$G_x = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} * A \quad , \quad G_y = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} * A$$

for the Roberts case [92]. The magnitude of the edge, G, can be approximated as $(|G_x| + |G_y|)/2^n$, where n is a scaling factor. As soon as the magnitude is evaluated, a thresholding stage (threshold value depends on the specific context of application) compares the magnitude with a desired value, to determine whether the edge is present or not.

Roberts edge detector is the simplest gradient-based edge detector, due to the reduced convolution kernel matrix. It requires a bit less computation, resulting in limited resource footprint when implemented in hardware, with respect to Sobel [22]. However, this simplicity is paid in terms of detection effectiveness and noise robustness: Roberts is able to detect a smaller set of edges [15], and it is more sensitive to noise than Sobel [106]. Due to these characteristics, implementing both Sobel and Roberts edge detectors on the same device can be useful to achieve non-functional adaptivity. Sobel and Roberts kernels may correspond to different working points, each featuring a different tradeoff in terms of performance and detection power: Roberts constitutes the fastest, but less accurate detection; while Sobel pushes detection quality at the price of a slower execution. Such kind of changing behaviours could be suitable for CPS contexts where the system has to monitor the environment, the user demand and its internal state to promptly adapt the exhibited behaviour. For example, if the battery level of the system is lower than a certain threshold, it can switch from a more consuming detector, Sobel, to a less power-hungry one, Roberts. In the same way, if the incoming images are not so noisy, a low quality detector, Roberts, can be sufficient and more energy efficient; while, if the image source is noisy, Sobel, is preferable. Note that, such approach can also be exploited to achieve functional adaptation when Sobel and Roberts have to be adopted in the same application, such as for airport runway tracking on infra-red images [126].

7.3.2 Designs Under Tests

In the presented design flow, the applications to be implemented have to be modelled as dataflow graphs. In the adopted use-case, Sobel and Roberts have been described in CAPH dataflow language [99]. Figure 7.10 depicts simplified graphs of the dataflows representing Sobel and Roberts kernels, according to the algorithm described in Section 7.3.1. To compute convolution between the mask and the input image, the convolution actors need to receive the proper numbers of data. For this reason *line buffer* actors are adopted to store previous rows of the image, while *delay* actors are in charge of memorizing one previous pixel within a row. In Figure 7.10 the position within the edge detection kernel matrix related to each pixel coming from the input (*in pel*), a *line buffer* or a *delay* actor is highlighted (e.g. the pixel coming from *line buffer* in Roberts, that correspond to line 1 column 0 of the Roberts kernel matrices, will be multiplied by -1 in *roberts y* convolution actor). As expected by formula $(|G_x| + |G_y|)/2^n$, actor *abs sum* finalizes the magnitude computation step, by summing up the absolute values of the horizontal and vertical gradients and by right-shifting the result for a given scaling factor n (in this case no scaling is performed, $n = 0$). Lastly, the thresholding actor *thr* sets to the maximum pixel value (255) all the magnitudes that are above a certain threshold (it has been fixed to 80), while setting to 0 the others.

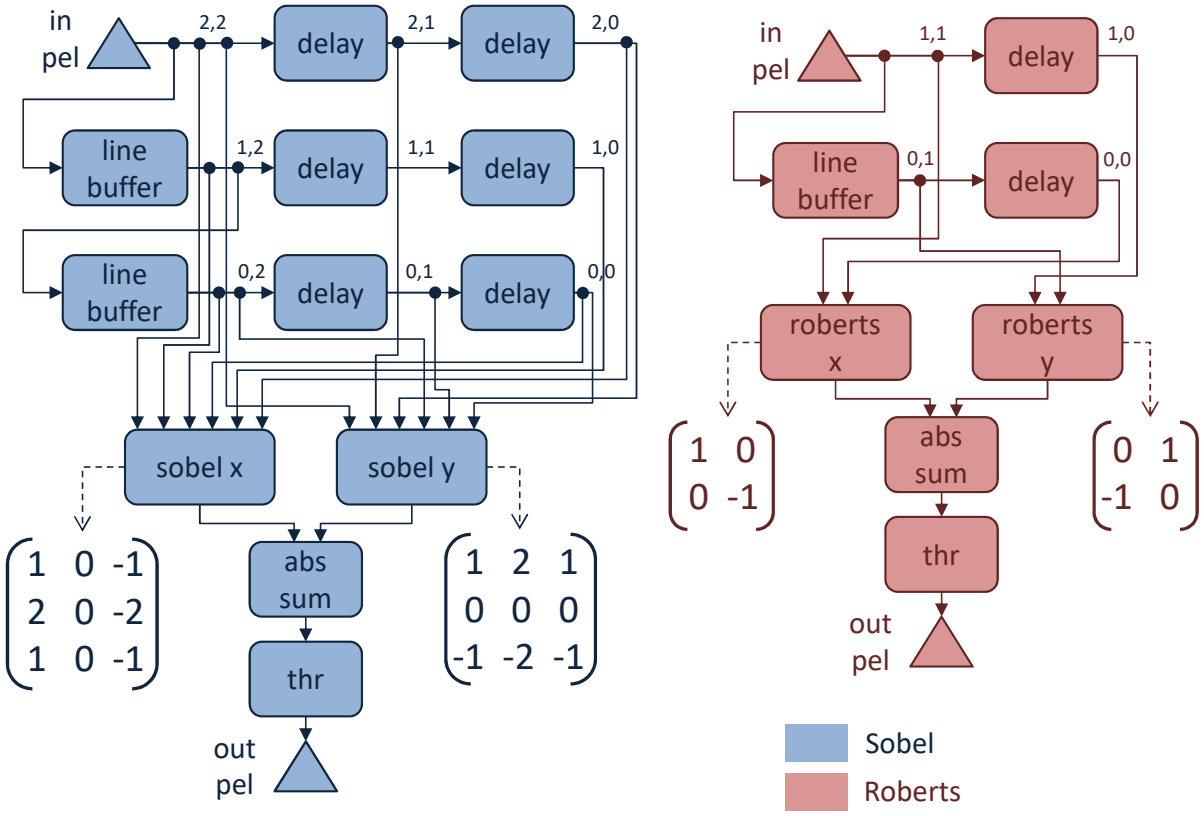


Figure 7.10: Simplified dataflow graphs implementing the Sobel and the Roberts edge detection computational kernels.

From these two dataflow models, three different computing cores can be derived: the *Sobel* kernel, the *Roberts* kernel and the *reconfigurable* kernel. The *Sobel* and *Roberts* kernels are directly derived from their dataflow models through a 1:1 mapping of the actor to hardware modules. The *reconfigurable* kernel is the result of *Sobel* and *Roberts* merging. In this *reconfigurable* kernel the actors in common between the two Sobel and Roberts functionalities are related to the magnitude computation step (*line buffer*, *delay* and *abs sum*) and to the thresholding step (*thr*). The convolutional actors, that are algorithm-specific, do not show any overlapping, so they are not shareable. Summarizing, the standalone *Sobel* and *Roberts* kernels involve respectively 14 and 7 actors, while *reconfigurable* kernel has 14 *SBoxes* (for detail on the *SBoxes*, please see Chapter 3) and 18 overall actors, among which 5 are shared between the Sobel and Roberts datapaths, and 13 not. On the bases of the described *Sobel*, *Roberts* and *reconfigurable* kernels, three different designs are considered:

- *fine-grain* - a standard DPR-based ARTICo^3 architecture where *Sobel* and *Roberts* kernels can be freely instantiated within a number of slots going from 1 to 4.
- *coarse-grain* - an MDC-compliant reconfigurable accelerator built around the *reconfigurable* computational kernel, able to switch between Sobel and Roberts algorithms. Contrarily to ARTICo^3, MDC cannot tune the number of accelerators to be used in parallel at runtime, so that a worst case configuration with 4 accelerators in parallel is adopted.
- *multi-grain* - following the approach proposed in this work, this implementation is composed of heterogeneous MDC-generated *reconfigurable* kernels instantiated within the slots of the ARTICo^3 architecture. The number of slots ranges from 1 to 4.

Please, note that designs are labelled according to the granularity level of the reconfiguration behind them (e.g. *fine* is used when single bit connections are changed, that is when DPR is enabled). Nevertheless, both ARTICo^3 and MDC architectures act on coarse areas of the FPGA. ARTICo^3 slots, from a chip occupation perspective, are larger than MDC CGR computing core, but they are reconfigured at bit level using DPR, which is why the standalone ARTICo^3 designs are labelled as *fine-grain*.

7.3.3 Experimental Results

To evaluate the proposed approach, experimental results obtained with the considered designs under test are hereafter discussed. The target device for all the reported data is the Xilinx XC7Z020CLG484-1 available on the adopted custom Zynq-7000 board, and the operating frequency has been set to 100 MHz. Note that energy numbers come from real on-board power measurements during execution. For the evaluation of the designs, several metrics will be taken into account:

- LUT, FF - resource occupancy of the design (for all its possible configurations) within the targeted device;
- fps - frames per second of the design (for all its possible configurations) while processing images;
- E - energy consumption of the design (for all its possible configurations) while processing images;
- Reconfiguration overhead, evaluated in terms of:
  - B - memory footprint of the reconfiguration data;
  - T - time required for the reconfiguration phase;
  - E - energy consumption during the reconfiguration phase;
  - F - penalty on the achievable maximum operating frequency when reconfiguration is provided;

Processing Evaluation

Table 7.1 and Table 7.2 depict hardware processing results for each possible configuration of the different designs: *fine-grain*, *coarse-grain* and *multi-grain*. For each design several configurations, labelled as *Nkn*, are considered, where N is the number of parallel accelerators, k is the kind of of computational kernel of each accelerator (f for the standalone computing cores (fixed) or r for the reconfigurable computing core), n refers to the name of the implemented functionality (s for *Sobel* and r for *Roberts*). For instance, *4fs* is the fixed, meant as not reconfigurable, computing Sobel with 4 accelerators in parallel, while *2rr* is the reconfigurable accelerator, computing Roberts functionality, with 2 accelerators in parallel.

Table 7.1 depicts processing time, in frames per second (*fps*), for different image sizes and for different degrees of parallelism. With respect to the parallelism, both *fine-grain* and *multi-grain* designs show the same trend, increasing the processed *fps* up to 40% going from 1 to 4 parallel computing cores. It does not seem to be present an appreciable difference between *Sobel* and *Roberts* kernel execution in terms of *fps*. The size of the input images has also an impact on the results. The *fps* decreases as the image size increases, being lower than 1 for high resolution images (2048x2048 size). However, we can see as, for all of them, the degree of parallelism keeps the same fps increasing trend. This same trend can be seen when analysing energy consumption results. Figure 7.11 illustrates the trends considering the coarse-grain *reconfigurable Sobel*, for different image sizes, when the number of parallel cores changes.

Table 7.1: Experimental timing results, in frames per second, for all the configurations of the considered designs. Data in brackets show the percentage variation of the configurations with respect to the case where only one slot is exploited.

config	fps (for different image sizes)			
	256x256	**512x512**	**1024x1024**	**2048x2048**
fine-grain				
1fs	26.95	6.61	1.65	0.42
2fs	33.35 (+24%)	8.31 (+26%)	2.11 (+28%)	0.53 (+25%)
3fs	34.57 (+28%)	9.01 (+36%)	2.26 (+37%)	0.57 (+34%)
4fs	36.96 (+37%)	9.50 (+44%)	2.39 (+44%)	0.59 (+40%)
1fr	26.91	6.64	1.70	0.42
2fr	33.48 (+24%)	8.43 (+27%)	2.12 (+25%)	0.53 (+25%)
3fr	34.58 (+29%)	9.00 (+35%)	2.28 (+34%)	0.57 (+35%)
4fr	35.98 (+34%)	9.42 (+42%)	2.37 (+40%)	0.60 (+42%)
coarse-grain				
4rs	36.90	9.52	2.39	0.60
4rr	36.96	9.52	2.40	0.60
multi-grain				
1rs	26.85	6.74	1.69	0.42
2rs	33.19 (+24%)	8.38 (+24%)	2.12 (+25%)	0.53 (+25%)
3rs	34.45 (+28%)	8.96 (+33%)	2.23 (+32%)	0.57 (+35%)
4rs	36.90 (+37%)	9.52 (+41%)	2.39 (+41%)	0.60 (+42%)
1rr	26.90	6.62	1.70	0.42
2rr	33.06 (+23%)	8.42 (+27%)	2.12 (+24%)	0.53 (+25%)
3rr	34.37 (+28%)	8.10 (+36%)	2.28 (+34%)	0.57 (+34%)
4rr	36.96 (+37%)	9.52 (+44%)	2.40 (+41%)	0.60 (+42%)

It is evident that the performance does not increase linearly, as it would be expected. The main reason behind this is the communication overhead introduced by the ARTICo^3 architecture, and its predominance with respect to the real computing time (less than 10% of the overall time per block). To comply with the required data-parallel execution model, the image processing application requires partitioning the total workload in a given set of data-independent sub-workloads that are executed in rounds. The size of the local workload has been fixed to process 32x32 image blocks, and the *Runtime Library* dispatches processing rounds in as many slots as available. Adopting more complex computational kernels or over-clocking data transfers could highlight differences between the two kernels and executions with different number of slots.

Table 7.2 shows the measured energy consumption for all configurations. As occurs for performance, the energy efficiency increases when using more accelerators (power consumption increasing is less significant than the saving in terms of execution time), reaching reduction values of up to 27%. Comparing the *fine-grain* and the *multi-grain* approaches, it is possible to see that energy consumption is similar. This means that, in an application scenario in which both edge detectors need to be present, the *multi-grain* accelerator provides a solution that is more energy efficient, since both datapaths are already available in the accelerator core logic (otherwise, two *fine-grain* accelerators would need to be present in the FPGA, or switched using DPR, which also adds an energy overhead

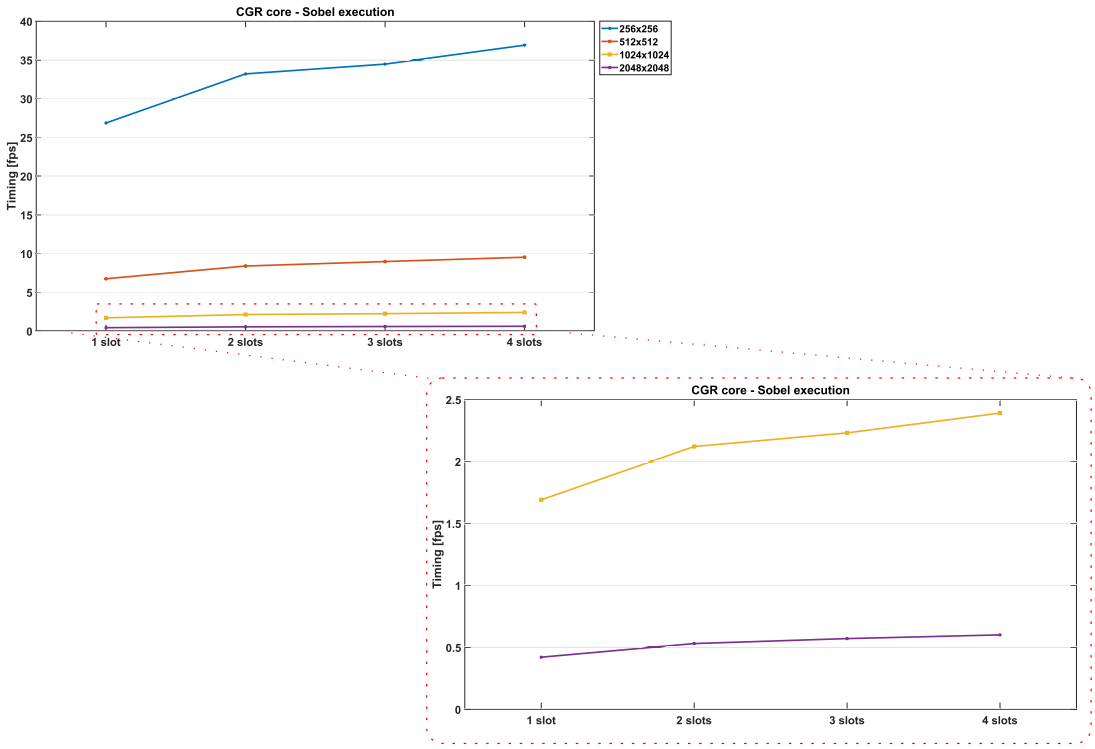


Figure 7.11

as discussed in the following section). Note that, as for the *fps* metric, in terms of energy there is not a real difference between *Sobel* and *Roberts* kernel execution, meaning that CGR, in this very simple case, is not able to provide non-functional (fps/energy driven) adaptivity, but only a functional (algorithm driven) one. The predominance of DMA transfers is then playing a role also in terms of energy, flattening differences between the two kernels and among the different slot configurations.

Reconfiguration Granularity Analysis

Table 7.3 depicts overhead results of both *fine-grain* and *coarse-grain* reconfigurations, in terms of memory footprint (size [B]), timing (time [ms]) and energy consumption (energy [mJ]). Generally speaking the reconfiguration overhead of *coarse-grain* designs is much lower than the overhead of *fine-grain* ones. this result is not un-expected, indeed reconfiguration in MDC computing cores is enabled by simply writing a single configuration register for each accelerator. In ARTICo^3 architecture, the DPR-enabled reconfiguration requires downloading a new bitstream for the whole slot. In the presented scenario, the CGR approach of four parallel accelerators requires around 1000 times less reconfiguration time (0.09 ms versus 106.14 ms) and around 850 times less reconfiguration energy consumption (0.11 mJ versus 94.11 mJ) with respect to fine-grain reconfiguration of four slots. If we take into account an application example in which it is necessary to switch N times among the *Sobel* and *Roberts* edge detection algorithms, the reconfiguration cost for the four *coarse-grain* parallel accelerators would be $N \times 0.09ms$ and $N \times 0.11mJ$, while the cost for reconfiguring the four parallel slot with the different kernels would be $N \times 106.14ms$ and $N \times 94.11mJ$.

The price to be paid for this advantage in terms of timing and energy is that *coarse-grain* designs are less flexible. Only a limited set of functionalities are implemented in a CGR computing core, and it is only possible to multiplex them in time. Furthermore, an infrastructure providing only CGR computing cores does not offer the possibility of changing the parallelism degree or re-writing completely the functionality with a bitstream not originally considered at design time. A *multi-grain* adaptive de-

Table 7.2: Experimental energy results in [mJ] for all the configurations of the considered designs. *Coming from real on-board power measurements. Data in brackets show the percentage variation of the configurations with respect to the case with only one slot.

config	energy* [mJ] (for different image sizes)			
	256x256	**512x512**	**1024x1024**	**2048x2048**
fine-grain				
1fs	33.47	130.32	532.13	2132.95
2fs	28.61 (-14%)	107.27 (-17%)	437.37 (-17%)	1756.8 (-17%)
3fs	26.76 (-20%)	102.38 (-21%)	406.44 (-23%)	1628.57 (-23%)
4fs	25.26 (-24%)	99.01 (-24%)	397.11 (-25%)	1562.94 (-26%)
1fr	32.36	131.76	526.17	2106.31
2fr	27.23 (-15%)	106.58 (-19%)	429.57 (-18%)	1729.04 (-17%)
3fr	26.93 (-16%)	100.42 (-23%)	402.62 (-23%)	1603.06 (-23%)
4fr	25.29 (-21%)	96.08 (-27%)	387.19 (-26%)	1537 (-27%)
coarse-grain				
4rs	28.44	97.89	397.4	1578.09
4rr	25.42	97.22	399.73	1570.45
multi-grain				
1rs	33.04	133.66	529.74	2136.26
2rs	26.98 (-18%)	109.63 (-17%)	440.38 (-16%)	1751.99 (-17%)
3rs	27.82 (-15%)	102.26 (-23%)	412.56 (-22%)	1636.79 (-23%)
4rs	28.44 (-13%)	97.89 (-26%)	397.4 (-24%)	1578.09 (-26%)
1rr	32.11	131.02	527.9	2128.85
2rr	27.01 (-15%)	107.24 (-18%)	435.99 (-17%)	1755.22 (-17%)
3rr	26.36 (-17%)	103.15 (-21%)	406.78 (-22%)	1631.55 (-23%)
4rr	25.42 (-20%)	97.22 (-25%)	399.73 (-24%)	1570.45 (-26%)

Table 7.3: Reconfiguration overhead. *N is the number of parallel accelerators (slots). **Real on-board power measurements.

design	**config***	**size [B]**	**time [ms]**	**energy** [mJ]**
fine-grain	1	858k	16.42	15.18
fine-grain	2	1715k	47.62	41.91
fine-grain	3	2573k	75.95	67.1
fine-grain	4	3430k	106.14	94.11
coarse-grain	4	2	0.09	0.11

sign combines the best of both, offering a fast low-power consuming reconfiguration, when changing among a set of functionalities is needed, and the performance trade-off (throughput versus energy consumption) given by the flexible parallelism offered by the DPR-based ARTICo^3 architecture. Also, with DPR is possible to replace the CGR computing core with another one, when a different set of functionalities is necessary. For the considered proof of concept involving *Sobel* and *Roberts* algo-

rithms, MDC-based reconfiguration can be exploited to change the functionality of each slot in a lightweight manner, while the DPR approach behind ARTICo^3 takes care of fps/energy by playing with the adopted number of slots.

Impact of Coarse-Grain Reconfiguration

As already described in Chapter 3, reconfiguration in MDC designs is guaranteed by the insertion of multiplexers (*SBoxes*) in the crossroads of common paths of data. All the resources necessary to the computation of each functionality plus the SBoxes are present on the substrate. This overhead is clearly visible in Table 7.4, where *Sobel* and *Roberts* occupy respectively 10% and 55% less LUT and FF than the *reconfigurable* kernel. Comparing the *reconfigurable* kernel resources utilization with the two isolated kernels instantiated in parallel (see the *Sobel+Roberts* row in Table 7.4), it is clear as the *reconfigurable* kernel saves more than 20% of resources, meaning that sharing actors is convenient and that the *SBoxes* overhead is affordable, according to the considered metric.

Table 7.4: Coarse-grain reconfiguration overhead (affecting *coarse-grain* and *multi-grain* designs in Section 7.3.1). In brackets percentages of variation of each metric wrt CGR design.

kernel	**resources (@100 MHz)**		**Fmax [MHz]**
	LUT	**FF**	
reconfigurable	2225	2360	108.39
Sobel	1817 (-18%)	2076 (-12%)	113.92 (+5%)
Roberts	922 (-59%)	1048 (-56%)	170.13 (+57%)
Sobel+Roberts	2739 (+23%)	3124 (+32%)	113.92 (+5%)

However, the insertion of the SBoxes has impact not only on the resource utilization, but it can also negatively affect the operating frequency. *SBoxes* are fully combinatorial; therefore, their presence may lengthen the critical path of the system, thus lowering the highest achievable frequency. In this case, non-reconfigurable *Sobel* and *Roberts* kernels support respectively 5% and 57% higher operating frequency than the *reconfigurable* one. For the parallel *Sobel+Roberts* solution, if a worst case frequency has to be selected (that is also the case of *reconfigurable* kernel), the maximum frequency estimated for *Sobel* need to be considered.

Usage of the Proposed Flow

Previous discussions demonstrated the effectiveness of multi-grain reconfiguration, showing the advantage, in terms of timing and energy consumption, of mixing ARTICo^3 and MDC reconfiguration approaches. It is important to highlight also the effectiveness of the proposed flow, in terms of design time and effort. The Integrated MDC-ARTICo^3 toolchain automatically maps different input specifications in one CGR datapath compliant with the DPR-based ARTICo^3 slots, speeding-up the design of multi-grain systems. Users only need to define the applications behaviour through abstract high level input dataflow specifications.

The effort of designing the dataflow specifications is application specific and cannot be evaluated. However, it is necessary to highlight as the usage of dataflow specifications allows exploitation of HLS dataflow-to-hardware tools (such as the adopted CAPH [99]), which not only speed the design process up by automating HDL generation, but allow developers that are not expert in hardware design to adopt the proposed design flow.

Furthermore, the proposed toolchain facilitates the management of the generated multi-grain system, since the *Runtime Library* of the ARTICo^3 architecture is naturally capable of managing the application execution and computation offloading to the hardware accelerators also when these are CGR accelerators.

7.4 Chapter Remarks

The combination of coarse-grain (time-multiplexing of available datapaths) and fine-grain (time-multiplexing of FPGA resources) approaches can be used to enable advanced functional and non-functional adaptivity support in CPS. The presented automated toolchain integrates the MDC tool with the ARTICo^3 framework to support the development, from specification down to implementation, of multi-grain reconfigurable systems, speeding up the design process and facilitating their deployment and runtime management.

The proposed reconfiguration infrastructure has been evaluated on the use-case involving two edge detection kernels. Experimental results of this proof-of-concept test case demonstrated the potential of the approach in terms of FPGA resources, timing and energy efficiency. Drawbacks and strengths of the different reconfiguration granularities have been highlighted, and the advantages of leveraging on a multi-grain architecture have been revealed. The proposed methodology can be particularly useful in CPS contexts, where variability is common due to the involvement of user (e.g., changing image resolution), environment (e.g., speed of the objects whose edges have to be detected) or system (e.g. remnant battery) requirements. With the edge detection proof of concept, it has been shown a limited set of the adaptivity potentials for the proposed approach. In the considered scenario, MDC lightweight adaptation provided functional adaptation but, in cases where the implemented computational kernels differ substantially, it could be used to provide also non-functional adaptation with performance/energy tradeoff [97]. Similarly, ARTICo^3-based adaptivity could achieve better results with scalability on more complex applications or could provide fault tolerance [93].

Future steps for this research involve a deeper validation of the proposed architecture with the Planetary Exploration (*PE*) CERBERO project use-case. The *PE* use-case aims at assessing a new technology for computing purposes in Space applications, where robustness to faults has to be guaranteed and self-monitoring and self-healing capabilities are meant to be supported to prevent and overcome failures caused by radiation and harsh environmental conditions. The final demonstrator is the controller of a robotic arm implemented over a FPGA device. The multi-grain reconfiguration presented in this Chapter will allow the controller to switch, according to requirements, among several algorithm (differing for time convergence and smoothness) for controlling the arm movement. Parallel scalability will guarantee performance flexibility and fault tolerance. Self-monitoring and self-adaptive processing capabilities will be given by integrating the MDC-ARTICo^3 toolchain with other tools belonging to CERBERO partnership, to provide the advanced runtime adaptation of the multi-grain architecture according to data collected by monitoring the performance.

List of Publications Related to the Chapter

Conference papers

- Tiziana FANNI, Alfonso Rodríguez, Carlo Sau, Leonardo Suriano, Francesca Palumbo, Luigi Raffo and Eduardo de la Torre, *Multi-Grain Reconfiguration for Advanced Adaptivity in Cyber-Physical Systems*. 2018 International Conference on ReConFigurable Computing and FPGAs (ReConFig). December 2018. Proceedings in press.

- Alfonso Rodríguez and Tiziana FANNI, *DEMO: Multi-Grain Adaptivity in Cyber-Physical Systems*. Special Session on Energy Efficient Cyber Physical Systems held at the 30th International Conference on Microelectronics (ICM).December 2018. Proceedings in press.

Other scientific papers

- Carlo Sau, Tiziana FANNI, Luigi Raffo and Francesca Palumbo *Self-adaptation of Cyber-Physical Systems*, at the 2018 Riunione Annuale dell'Associazione Società Italiana di Elettronica (SIE – ex GE), Napoli (Italy) June 2018.
- Tiziana FANNI, Alfonso Rodríguez, Carlo Sau, Francesca Palumbo, Luigi Raffo and Eduardo de la Torre, *Providing Advanced Adaptivity in Cyber-Physical Systems with Multi-Grain Reconfiguration*, at the 2018 Italian Workshop on Embedded Systems (IWES), Siena (Italy), September 2018.

Chapter 8

Concluding remarks

Reconfigurable computing seems to be a good solution for the challenging world of Cyber-Physical Systems (CPS). It allows devices to achieve a tradeoff between flexibility, performance and power consumption. In particular, coarse-grain reconfigurability guarantees quick functionality switches even if flexibility, that is number of supported functionalities, is limited and fixed at design time. For these characteristics, they result very convenient in real-time application specific contexts. However, in such a kind of systems, resources that are not involved in the current computation can easily lead to a waste of energy, so that advanced power management is mandatory. The main drawback common to low power reconfigurable architectures is related to the typically huge cost required by their efficient design and management.

The objectives of this thesis work were mainly related to the development of methodologies for aiding designers in the complex and time consuming process of deploying low power reconfigurable architectures, exploiting the dataflow models of computation. For achievement of these objectives, power saving strategies for ASIC and FPGA have been studied and deeply analysed. When possible, the presented approaches have been integrated in the Multi-Dataflow Composer (MDC), a dataflow to hardware framework that starting from the specification of the desired functionalities (modelled as dataflow networks), can derive the RTL description of the corresponding coarse grained reconfigurable substrate able to implement all the input functionalities. Figure 8.1 illustrates the development timeline of MDC, highlighting the extensions and modifications carried out during this thesis work.

Regarding the application of power saving techniques for ASIC design, the MDC *Power Manager* has been extended to support techniques for the management and saving of the static power. In particular, the capability of MDC of applying automatically the clock gating methodology (which acts only on dynamic power) has been extended to implement also a coarse-grain power gating strategy for ASIC designs (4). To apply the power gating, firstly the MDC *Logic Regions Identification*[1] algorithm (see algorithm 1 in Appendix A) has been modified to include also the switching modules (SBoxes) of the coarse-grain reconfigurable (CGR) system (see algorithm 3 in Appendix B) that, being combinational, were not included in power gating related analysis. Then, to give the information about the power specification to the synthesizer, MDC has been extended to generate also a common power format (CPF) file. MDC *Power Manager* has also been further improved to include a power modelling flow leading the designer to an optimal power saving strategy. This methodology is based on static and dynamic power estimation models that, considering separately each logic region in the CGR design, are capable of determining the overhead of clock gating and power gating on the basis of the functional, technological and architectural parameters of the baseline CGR system.

[1] A logic region is a set of processing elements always active or inactive together.

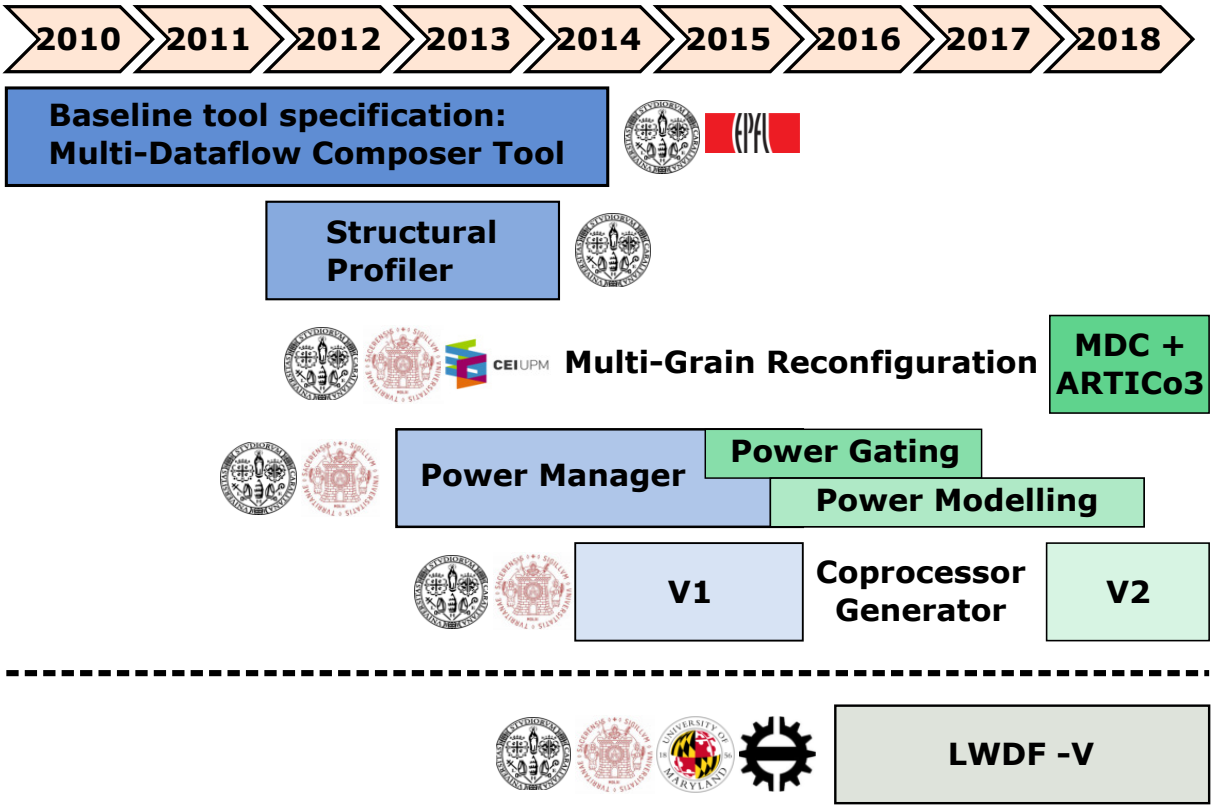


Figure 8.1: Multi-Dataflow Composer Tool - Development Timeline at 2018

Given that MDC is able to apply clock gating for both ASIC and FPGA, and that in FPGA context power gating is not applicable (as discussed more in detail in Chapter 4), different ways to deal with energy consumption have been explored. In particular, the combination of CGR systems with the dynamic partial reconfiguration has been proposed, to enable advanced functional and non-functional adaptivity support in CPS, since different tradeoffs between performance, flexibility and energy consumptioncan be provided. MDC has been integrated with ARTICo^3, a DPR-enable framework that exploits a multi-accelerators scheme, to constitute an automated toolchain for the development and management of multi-grain reconfigurable systems. This integration required to align the MDC with the Vivado Design Environment, thus it has been necessary to provide a new version of the *Coprocessor Generator* (Coprocessor Generator V2 in Figure 8.1).

Sometimes, system developers need to deal with models and design approaches that are not mature yet. In these cases, ad-hoc methods are required to explore different design optimisation and power management techniques. At this purpose, in the thesis work, a compact set of retargetable API for lightweight dataflow (LWDF) and implementation using HDL has been studied. During the thesis the natural integration of power management techniques within such APIs has been proposed. In particular it has been presented LIDE-V, an extension of the lightweight dataflow environment (LIDE) that provides support for Verilog-based implementation of the LWDF APIs, along with associated libraries of dataflow actor and edge implementations. LIDE-V facilitates the design of experimentation with alternative implementations of a given LWDF-V model to reveal important insights into system-level tradeoffs, and perform multidimensional design optimization. This methodology has not been automated yet but it promises to be a valuable tool in the design of low power dataflow-based systems, and it could be extended for the deployment of low power CGR systems. The study of LWDF is illustrated in the bottom part of Figure 8.1.

8.1 Future Works

The methodologies and approaches for power management and their integrations into the MDC tool that have been presented during this thesis work could be enhanced under different aspects. Dealing with ASIC systems (Chapter 4 and Chapter 5), a follow up of this research includes improving the power estimation model, considering also the contribution of the interconnection which currently is not addressed. Furthermore, power switches overhead is not considered yet; these sleep transistors are inserted only during place and route design steps, and a way to estimate in advance how many switches are going to be inserted for each logic region has not been explored yet. Indeed the number of switches has effect on the rush currents during power-up transitions, and on the power-down/up timing thus also these aspects need to be included in a *Power Modelling* methodology able to identify the best power saving strategy .

Regarding the power management in FPGA, involving the multi-grain approach (Chapter 7), next steps involve the integration of the MDC-ARTICo^3 tools into an automate framework able to offer support at design time and at runtime for self-adaptation in CPS. In particular the multi-grain architecture is going to be instrumented with hardware monitors, to give information of the system performance to a runtime manager that can make decisions on the reconfiguration. The complete self-adaptation framework is going to be demonstrated on the Planetary Exploration (*PE*) CERBERO project use-case. The *PE* use-case final demonstrator is a controller, implemented over an FPGA, for a robotic arm, to be used on a rover on the next space missions on Mars. The multi-grain reconfiguration presented in this thesis will allow to guarantee fast reconfiguration among algorithms for controlling the arm, and robustness to failures caused by radiation and harsh environmental conditions.

Lastly, considering the exploration of system-level models for the application of power management techniques, next steps include the development of a design methodology, based on the application of LWDF (see Chapter 6). This design methodology should enable experimentation across different levels of abstraction throughout the design process, assisting designers of signal processing systems in exploring complex design alternatives that span multiple implementation scales, platform types, and dataflow modelling techniques. This would allow designers to experiment productively and iterate rapidly on complex combinations of design options, enabling effective experimentation with hardware/software design tradeoffs, as well as tradeoffs involving performance, resource utilization, and power consumption.

Bibliography

[1] [cited at p. 10, 113]

[2] OpenCL - the open standard for parallel programming of heterogeneous systems. In *https://www.khronos.org/opencl/*. [cited at p. 5]

[3] M. Rückauer A. Thomas and J. Becker. Honeycomb: An application-driven online adaptive reconfigurable hardware architecture. *Int. J. Reconfig. Comp.*, 2012:17 pages, 2012. [cited at p. 113]

[4] A. Agarwal, S.K. Mathew, S.K. Hsu, M.A. Anders, H. Kaul, F. Sheikh, R. Ramanarayanan, S. Srinivasan, R. Krishnamurthy, and S. Borkar. A 320mv-to-1.2v on-die fine-grained reconfigurable fabric for dsp/media accelerators in 32nm cmos. volume 53, pages 328–329, 2010. cited By 19. [cited at p. 9, 11]

[5] Altera. *FPGA Coprocessing Evolution: Sustained Performance Approaches Peak Performance*, 2009. [cited at p. 8]

[6] Altera. *Increasing Design Functionality with Partial and Dynamic Reconfiguration in 28-nm FPGAs*, 2010. [cited at p. 8]

[7] M. Arora, S. Manne, Y. Eckert, I. Paul, N. Jayasena, and D. M. Tullsen. A comparison of core power gating strategies implemented in modern hardware. In *ACM SIGMETRICS / International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS '14*, pages 559–560, 2014. [cited at p. 38]

[8] Krste Asanović, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006. [cited at p. 114]

[9] E. Bezati, S. Casale-Brunet, M. Mattavelli, and J. W. Janneck. Clock-gating of streaming applications for energy efficient implementations on fpgas. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 36(4):699–703, April 2017. [cited at p. 38, 90]

[10] E. Bezati, M. Mattavelli, and J.W. Janneck. High-level synthesis of dataflow programs for signal processing systems. In *Image and Signal Processing and Analysis (ISPA), 2013 8th International Symposium on*, pages 750–754, September 2013. [cited at p. 15]

[11] B. Bhattacharya and S. S. Bhattacharyya. Parameterized dataflow modeling for dsp systems. *IEEE Transactions on Signal Processing*, 49(10):2408–2421, Oct 2001. [cited at p. 14]

[12] S. S. Bhattacharyya, E. Deprettere, R. Leupers, and J. Takala, editors. *Handbook of Signal Processing Systems*. Springer, second edition, 2013. ISBN: 978-1-4614-6858-5 (Print); 978-1-4614-6859-2 (Online). [cited at p. 11, 13]

[13] S. C. Brunet, E. Bezati, C. Alberti, M. Mattavelli, E. Amaldi, and J. W. Janneck. Multi-clock domain optimization for reconfigurable architectures in high-level dataflow applications. In *2013 Asilomar Conference on Signals, Systems and Computers*, pages 1796–1800, Nov 2013. [cited at p. 90]

[14] J.T Buck. *Scheduling dynamic dataflow graphs with bounded memory using the token flow model*. PhD thesis, Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, 1993. [cited at p. 13]

[15] J. Burnham, J. Hardy, and K. Meadors. Comparison of the roberts, sobel, robinson, canny, and hough image detection algorithms. In *MS State DSP Conf.*, 1997. [cited at p. 127]

[16] Cadence®. *Low Power in Encounter®RTL Compiler, Product Version 14.1*, July 2014. [cited at p. 62]

[17] Cadence®. *Using Encounter®RTL Compiler, Product Version 14.1*, July 2014. [cited at p. 38]

[18] Cadence®. Genus synthesis solution, 2018. https://www.cadence.com/content/cadence-www/global/en_US/home/tools/digital-design-and-signoff/synthesis/genus-synthesis-solution.html, 2018-08-28. [cited at p. 38]

[19] A. Cappelli, A. Lodi, M. Bocchi, C. Mucci, M. Innocenti, C. De Bartolomeis, L. Ciccarelli, R. Giansante, A. Deledda, F. Campi, M. Toma, and R. Guerrieri. Xisystem: a xirisc-based soc with a reconfigurable io module. In *Solid-State Circuits Conference, 2005. Digest of Technical Papers. ISSCC. 2005 IEEE International*, pages 196–593 Vol. 1, February 2005. [cited at p. 8]

[20] S.M. Carta, D. Pani, and L. Raffo. Reconfigurable coprocessor for multimedia application domain. *Journal of VLSI signal processing systems for signal, image and video technology*, 44(1):135–152, 2006. [cited at p. 19]

[21] S. Casale-Brunet, M. Mattavelli, and J.W. Janneck. Turnus: A design exploration framework for dataflow system design. In *Circuits and Systems (ISCAS), 2013 IEEE International Symposium on*, pages 654–654, May 2013. [cited at p. 15]

[22] G. N. Chaple, R. D. Daruwala, and M. S. Gofane. Comparisions of robert, prewitt, sobel operator based edge detection methods for real time uses on fpga. In *Int. Conf. on Technologies for Sustainable Development (ICTSD)*, 2015. [cited at p. 127]

[23] A. Chhabra, H. Rawat, M. Jain, P. Tessier, D. Pierredon, L. Bergher, and P. Kumar. FALPEM: framework for architectural-level power estimation and optimization for large memory subsystems. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 34(7):1138–1142, 2015. [cited at p. 60]

[24] RVC-CAL Community. Open RVC-CAL compiler (Orcc), 2018. http://orcc.sourceforge.net/, 2018-08-29. [cited at p. 15, 20]

[25] K. Compton and S. Hauck. Reconfigurable computing: A survey of systems and software. *ACM Comput. Surv.*, 34(2):171–210, June 2002. [cited at p. 7]

[26] J.W. Cooley and J.W. Tukey. An algorithm for the machine computation of complex fourier series, vol. 19. *Mathematics of Computation*, 1965. [cited at p. 73]

[27] C. E. Cummings. Simulation and synthesis techniques for asynchronous FIFO design. In *Proceedings of the Synopsys Users Group Conference*, 2002. [cited at p. 98]

[28] M. Danelutto, D. De Sensi, and M. Torquati. A power-aware, self-adaptive macro data flow framework. *Parallel Processing Letters*, 27(1):1–20, 2017. [cited at p. 90]

[29] K. Datta, A. Mukherjee, G. Cao, R. Tenneti, V. Vijendra Kumar Lakshmi, A. Ravindran, and B. S. Joshi. Casper: Embedding power estimation and hardware-controlled power management in a cycle-accurate micro-architecture simulation platform for many-core multi-threading heterogeneous processors. *Journal of Low Power Electronics and Applications*, 2(1):30–68, 2012. [cited at p. 60]

[30] E.R. Davies. Circularity - a new principle underlying the design of accurate edge orientation operators. *Image and Vision Computing*, 2(3):134–142, 1984. [cited at p. 125]

[31] J. B. Dennis. First version of a data flow procedure language. In *Programming Symposium, Proceedings Colloque Sur La Programmation*, pages 362–376, London, UK, UK, 1974. Springer-Verlag. [cited at p. 11]

[32] K. Desnos and J. Heulot. Pisdf: Parameterized & interfaced synchronous dataflow for mpsocs runtime reconfiguration. In *1st Workshop on MEthods and TOols for Dataflow PrOgramming (METODO)*, 2014. [cited at p. 14]

[33] C. M. Diniz, M. Shafique, S. Bampi, and J. Henkel. Run-time accelerator binding for tile-based mixed-grained reconfigurable architectures. In *Int. Conf. on Field Programmable Logic and Applications (FPL)*, 2014. [cited at p. 113]

[34] C. M. Diniz, M. Shafique, S. Bampi, and J. Henkel. A reconfigurable hardware architecture for fractional pixel interpolation in high efficiency video coding. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 34(2):238–251, Feb 2015. [cited at p. 85]

[35] J. Eker and J. Janneck. Cal language report. Technical report, Tech. Rep. ERL Technical Memo UCB/ERL, 2003. [cited at p. 13]

[36] J. Eker and J. W. Janneck. Dataflow programming in CAL — balancing expressiveness, analyzability, and implementability. In *Proceedings of the IEEE Asilomar Conference on Signals, Systems, and Computers*, pages 1120–1124, 2012. [cited at p. 90]

[37] S. Eyerman and L. Eeckhout. Fine-grained DVFS using on-chip regulators. *ACM Transactions on Architecture and Code Optimization (TACO)*, 8(1):1–24, 2011. [cited at p. 38]

[38] J. Fowers, J.Y. Kim, D. Burger, and S. Hauck. A scalable high-bandwidth architecture for lossless compression on fpgas. In *The 23rd IEEE International Symposium on Field-Programmable Custom Computing Machines*. IEEE - Institute of Electrical and Electronics Engineers, May 2015. [cited at p. 8]

[39] K. Gagarski, M. Petrov, M. Moiseev, and I. Klotchkov. Power specification, simulation and verification of systemc designs. In *2016 IEEE East-West Design Test Symposium (EWDTS)*, pages 1–4, Oct 2016. [cited at p. 39]

[40] K. Gilles. The semantics of a simple language for parallel programming. *In Information Processing*, 74:471–475, 1974. [cited at p. 11]

[41] R.W. Hartenstein. Coarse grain reconfigurable architecture (embedded tutorial). In *Proceedings of ASP-DAC 2001, Asia and South Pacific Design Automation Conference 2001, January 30-February 2, 2001, Yokohama, Japan*, pages 564–570, 2001. [cited at p. 9, 10]

[42] R.W. Hartenstein. A decade of reconfigurable computing: a visionary retrospective. In *Proc. Design, Automation and Test in Europe (DATE'01)*, pages 642–649, 2001. [cited at p. 37]

[43] D. Helms, R. Eilers, M. Metzdorf, and W. Nebel. Leakage models for high-level power estimation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(8):1627–1639, Aug 2018. [cited at p. 60]

[44] S. Herbert and D. Marculescu. Analysis of dynamic voltage/frequency scaling in chip-multiprocessors. In *Proceedings of the 2007 international symposium on Low power electronics and design (ISLPED '07)*, pages 38–43, Aug 2007. [cited at p. 38]

[45] P. Heysters, G. Smit, and E. Molenkamp. A flexible and energy-efficient coarse-grained reconfigurable architecture for mobile systems. *The Journal of Supercomputing*, 26(3):283–308, 2003. [cited at p. 9, 11]

[46] S. Holmbacka, E. Nogues, M. Pelcat, S. Lafond, D. Menard, and J. Lilius. Energy-awareness and performance management with parallel dataflow applications. *Journal of Signal Processing Systems*, 87(1):33–48, Apr 2017. [cited at p. 90]

[47] Z. Hu, A. Buyuktosunoglu, V. Srinivasan, V. Zyuban, H. Jacobson, and P. Bose. Microarchitectural techniques for power gating of execution units. In *Proceedings of the 2004 International Symposium on Low Power Electronics and Design*, ISLPED '04, pages 32–37, New York, NY, USA, 2004. ACM. [cited at p. 85]

[48] H. Huttunen, F. Yancheshmeh, and K. Chen. Car type recognition with deep neural networks. *ArXiv e-prints*, 2016. arXiv:1602.07125v2, To appear in proceedings of IEEE Intelligent Vehicles Symposium 2016. [cited at p. 102]

[49] IEEE Standard for Design and Verification of Low-Power, Energy-Aware Electronic Systems. *IEEE Standard 1801-2015, UPF-2.0, Unified Power Format 2.0*, 2016. [cited at p. 38]

[50] Q. Inoue, K.and Zhao, Y. Okamoto, H. Yosho, M. Amagasaki, M. Iida, and T. Sueyoshi. A variable-grain logic cell and routing architecture for a reconfigurable ip core. *ACM Trans. Reconfigurable Technol. Syst.*, 4(1):5:1–5:24, December 2010. [cited at p. 10]

[51] Chiu J.-C., Chou Y.-L., and Lin R.-B. The multi-context reconfigurable processing unit for fine-grain computing. *Journal of Information Science and Engineering*, 24(3):965–979, 2008. cited By 4. [cited at p. 9, 11]

[52] S. M. A. H. Jafri, M. A. Tajammul, A. Hemani, K. Paul, J. Plosila, and H. Tenhunen. Energy-aware-task-parallelism for efficient dynamic voltage, and frequency scaling, in cgras. In *2013 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, pages 104–112, July 2013. [cited at p. 38]

[53] B. Jeff. Advances in big. little technology for power and energy savings. *ARM White Paper*, 2012. [cited at p. 38]

[54] K. Keutzer, S. Malik, R. Newton, J. Rabaey, and A. Sangiovanni-Vincentelli. System-level design: orthogonalization of concerns and platform-based design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19, December 2000. [cited at p. 92]

[55] C. Kohn. *Partial Reconfiguration of a Hardware Accelerator on Zynq-7000 All Programmable SoC Devices*. Xilinx, January 2013. [cited at p. 8]

[56] R. R. Kulkarni and S. Y. Kulkarni. Energy efficient implementation, power aware simulation and verification of 16-bit alu using unified power format standards. In *2014 International Conference on Advances in Electronics Computers and Communications*, pages 1–6, Oct 2014. [cited at p. 38]

[57] V.V. Kumar and J. Lach. Highly flexible multimode digital signal processing systems using adaptable components and controllers. *EURASIP Journal on Applied Signal Processing*, 2006:73–73, 2006. [cited at p. 19]

[58] E.A. Lee and D.G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, September 1987. [cited at p. 13]

[59] E.A. Lee and T.M. Parks. Dataflow process networks. *Proceedings of the IEEE*, 83(5):773–801, 1995. [cited at p. 11]

[60] S. Li, J. Ho Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi. The mcpat framework for multicore and manycore architectures: Simultaneously modeling power, area, and timing. *TACO*, 10(1):5, 2013. [cited at p. 60]

[61] X. Li, K. Duraisamy, J. Baylon, T. Majumder, G. Wei, P. Bogdan, D. Heo, and P. P. Pande. A reconfigurable wireless noc for large scale microbiome community analysis. *IEEE Transactions on Computers*, 66(10):1653–1666, Oct 2017. [cited at p. 113]

[62] S. Lin et al. Parameterized sets of dataflow modes and their application to implementation of cognitive radio systems. *Journal of Signal Processing Systems*, 80(1):3–18, July 2015. [cited at p. 91]

[63] S. Lin, Y. Liu, W. Plishker, and S. S. Bhattacharyya. A design framework for mapping vectorized synchronous dataflow graphs onto CPU–GPU platforms. In *Proceedings of the International Workshop on Software and Compilers for Embedded Systems*, pages 20–29, Sankt Goar, Germany, May 2016. [cited at p. 91]

[64] L. Liu, Z. Li, C. Yang, C. Deng, S. Yin, and S. Wei. Hrea: An energy-efficient embedded dynamically reconfigurable fabric for 13-dwarfs processing. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 65(3):381–385, March 2018. [cited at p. 114]

[65] M. Lombardo, J. Camarero, J. Valverde, J. Portilla, E. de la Torre, and T. Riesgo. Power management techniques in an FPGA-based WSN node for high performance applications. In *7th International Workshop on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC)*, pages 1–8, July 2012. [cited at p. 111, 115]

[66] J. Lopes, D. Sousa, and J. C. Ferreira. Evaluation of cgra architecture for real-time processing of biological signals on wearable devices. In *2017 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–7, Dec 2017. [cited at p. 38]

[67] D. Macko. Contribution to automated generating of system power-management specification. In *2018 IEEE 21st International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS)*, pages 27–32, April 2018. [cited at p. 39]

[68] D. Madroñal, A. Morvan, R. Lazcano, R. Salvador, K. Desnos, E. Juárez, and C. Sanz. Automatic instrumentation of dataflow applications using papi. In *Proceedings of the 15th ACM International Conference on Computing Frontiers*, CF '18, pages 232–235, New York, NY, USA, 2018. ACM. [cited at p. 90]

[69] M. Masin, F. Palumbo, H. Myrhaug, J. A. de Oliveira Filho, M. Pastena, M. Pelcat, L. Raffo, F. Regazzoni, A. A. Sanchez, A. Toffetti, E. de la Torre, and K. Zedda. Cross-layer design of reconfigurable cyber-physical systems. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, pages 740–745, March 2017. [cited at p. 111]

[70] J. McAllister, R. Woods, R. Walke, and D. Reilly. Synthesis and high level optimisation of multi-dimensional dataflow actor networks on FPGA. In *Proceedings of the IEEE Workshop on Signal Processing Systems*, 2004. [cited at p. 15]

[71] N. Moreano, G. Araujo, Zhining Huang, and S. Malik. Datapath merging and interconnection sharing for reconfigurable architectures. In *System Synthesis, 2002. 15th International Symposium on*, pages 38–43, October 2002. [cited at p. 20]

[72] M. Musab and S. Yellampalli. Study and implementation of multi-vdd power reduction technique. In *2015 International Conference on Computer Communication and Informatics (ICCCI)*, pages 1–4, Jan 2015. [cited at p. 38]

[73] N. Nasirian, R. Soosahabi, and M. A. Bayoumi. Probabilistic analysis of power-gating in network-on-chip routers. *IEEE Transactions on Circuits and Systems II: Express Briefs*, pages 1–1, 2018. [cited at p. 60]

[74] Y. Nasser, J.C. Prevotet, and M. Hélard. Power Modeling for Fast Power Estimation on FPGA. working paper or preprint, https://hal.archives-ouvertes.fr/hal-01695867, 2018-08-28, February 2018. [cited at p. 60]

[75] J. F. Nezan, N. Siret, M. Wipliez, F. Palumbo, and L. Raffo. Multi-purpose systems: A novel dataflow-based generation and mapping strategy. In *IEEE Symposium on Circuits and Systems*, 2012. [cited at p. 15]

[76] A. Niedermeier, J. Kuper, and G. Smit. Dataflow-based reconfigurable architecture for streaming applications. In *System on Chip (SoC), 2012 International Symposium on*, pages 1–4, October 2012. [cited at p. 9, 10, 11]

[77] NVIDIA. Compute Unified Device Architecture (CUDA), 2018. https://www.nvidia.it/object/cuda-parallel-computing-it.html, 2018-08-29. [cited at p. 5]

[78] Mete Özbaltan and Nicolas Berthier. Exercising symbolic discrete control for designing low-power hardware circuits: an application to clock-gating. *IFAC-PapersOnLine*, 51(7):120 – 126, 2018. 14th IFAC Workshop on Discrete Event Systems WODES 2018. [cited at p. 38]

[79] G. Paim, L. M. G. Rocha, T. G. Alves, R. S. Ferreira, E. A. C. da Costa, and S. Bampi. A power-predictive environment for fast and power-aware asic-based fir filter design. In *2017 30th Symposium on Integrated Circuits and Systems Design (SBCCI)*, pages 168–173, Aug 2017. [cited at p. 60]

[80] F. Palumbo, D. Carta, N.and Pani, P. Meloni, and L. Raffo. The multi-dataflow composer tool: generation of on-the-fly reconfigurable platforms. *Journal of real-time image processing*, 9(1):233–249, 2014. [cited at p. 19, 20]

[81] F. Palumbo, C. Sau, and L. Raffo. DSE and Profiling of Multi-Context Coarse-Grained Reconfigurable Systems. In *International Symposium on Image and Signal Processing and Analysis*, pages 744–749, 2013. [cited at p. 23, 25]

[82] F. Palumbo, C. Sau, and L. Raffo. Coarse-grained reconfiguration: dataflow-based power management. volume 9, pages 36–48, 2014. [cited at p. 15, 25, 27, 38, 90]

[83] K. Paul, C. Dash, and M.S. Moghaddam. remorph: A runtime reconfigurable architecture. In *Digital System Design (DSD), 2012 15th Euromicro Conference on*, pages 26–33, September 2012. [cited at p. 10, 11]

[84] Massoud Pedram. Power minimization in ic design: principles and applications. *ACM Trans. Design Autom. Electr. Syst.*, 1:3–56, 1996. [cited at p. 38]

[85] M. Pelcat, K. Desnos, J. Heulot, C. Guy, J. Nezan, and S. Aridhi. Preesm: A dataflow-based rapid prototyping framework for simplifying multicore dsp programming. In *2014 6th European Embedded Design in Education and Research Conference (EDERC)*, pages 36–40, Sept 2014. [cited at p. 15, 90]

[86] J. Piat, S.S. Bhattacharyya, and M. Raulet. Interface-based hierarchy for synchronous data-flow graphs. In *Signal Processing Systems, 2009. SiPS 2009. IEEE Workshop on*, pages 145–150, October 2009. [cited at p. 14]

[87] W. Plishker, N. Sane, and S. S. Bhattacharyya. A generalized scheduling approach for dynamic dataflow applications. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, pages 111–116, 2009. [cited at p. 91]

[88] W. Plishker, N. Sane, M. Kiemb, K. Anand, and S. S. Bhattacharyya. Functional DIF for rapid prototyping. In *Proceedings of the International Symposium on Rapid System Prototyping*, pages 17–23, Monterey, California, June 2008. [cited at p. 14]

[89] W. Plishker, N. Sane, M. Kiemb, and S. S. Bhattacharyya. Heterogeneous design in functional DIF. In *Proceedings of the International Workshop on Systems, Architectures, Modeling, and Simulation*, pages 157–166, Samos, Greece, July 2008. [cited at p. 91, 98]

[90] Power Forward Initiative. *A Practical Guide to Low Power Design*, june 2009. [cited at p. 40, 84]

[91] A. Qamar, F. Bin Muslim, J. Iqbal, and L. Lavagno. Lp-hls: Automatic power-intent generation for high-level synthesis based hardware implementation flow. *Microprocessors and Microsystems*, 50:26 – 38, 2017. [cited at p. 39]

[92] L.G. Roberts et al. Machine perception of three dimensional solids, in optical and electro-optical information processing. *MIT press*, pages 159–197, 1965. [cited at p. 127]

[93] A. Rodríguez, J. Valverde, J. Portilla, A. Otero, T. Riesgo, and E. de la Torre. Fpga-based high-performance embedded systems for adaptive edge computing in cyber-physical systems: The artico3 framework. *Sensors*, 18(6), 2018. [cited at p. ix, 111, 115, 116, 117, 134]

[94] D. Rossi, A. Pullini, I. Loi, M. Gautschi, F. K. GÃ¼rkaynak, A. Teman, J. Constantin, A. Burg, I. Miro-Panades, E. BeignÃ¨, F. Clermidy, P. Flatresse, and L. Benini. Energy-efficient near-threshold parallel computing: The pulpv2 cluster. *IEEE Micro*, 37(5):20–31, September 2017. [cited at p. 38]

[95] I. Salvador, S. Remberto, M. Brox, and M. A. Ortiz. Software defined network controller: A neat solution administration for reconfigurable multi-core noc. In *2017 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–4, Dec 2017. [cited at p. 113]

[96] Synflow SAS. Synflow ide, 2018. http://www.synflow.com/, 2018-08-29. [cited at p. 15, 20]

[97] C. Sau, F. Palumbo, M. Pelcat, J. Heulot, E. Nogues, D. Menard, P. Meloni, and L. Raffo. Challenging the best hevc fractional pixel fpga interpolators with reconfigurable and multifrequency approximate computing. *IEEE Embedded Systems Letters*, 9(3):65–68, Sept 2017. [cited at p. 111, 115, 134]

[98] J. Sérot. The semantics of a purely functional graph notation system. In *Achten, P., Koopman, P.W.M., Morazán, M.T. (eds.) Draft Proceedings of the Ninth Symposium on Trends in Functional Programming, TFP 2008*. 2008. [cited at p. 15, 90]

[99] J. Sérot, F. Berry, and S. Ahmed. *CAPH: A Language for Implementing Stream-Processing Applications on FPGAs*, pages 201–224. Springer New York, 2013. [cited at p. 15, 20, 127, 133]

[100] J. Sérot, F. Berry, and title= Bourrasset, C. [cited at p. 15]

[101] M. Shafique, L. Bauer, and J. Henkel. Adaptive energy management for dynamically reconfigurable processors. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 33(1):50–63, 2014. [cited at p. 60]

[102] C. Shen, W. Plishker, and S. S. Bhattacharyya. Dataflow-based design and implementation of image processing applications. In L. Guan, Y. He, and S.-Y. Kung, editors, *Multimedia Image and Video Processing*, pages 609–629. CRC Press, second edition, 2012. [cited at p. 98]

[103] C. Shen, W. Plishker, H. Wu, and S. S. Bhattacharyya. A lightweight dataflow approach for design and implementation of SDR systems. In *Proceedings of the Wireless Innovation Conference and Product Exposition*, pages 640–645, 2010. [cited at p. 15, 90, 98]

[104] Y. Shyu, J. Lin, C. Lin, C. Huang, and S. Chang. An efficient and effective methodology to control turn-on sequence of power switches for power gating designs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(10):1730–1743, Oct 2016. [cited at p. 61]

[105] Silicon Integration Initiative. *Si2 Common Power Format Specification^{TM} - Version 2.1*, December 2014. [cited at p. 38]

[106] S. Singh and B. Singh. Effects of noise on various edge detection techniques. In *Int. Conf. on Computing for Sustainable Global Development (INDIACom)*, March 2015. [cited at p. 127]

[107] N. Siret, I. Sabry, J.F. Nezan, and M. Raulet. A codesign synthesis from an mpeg-4 decoder dataflow description. In *Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium on*, pages 1995–1998, May 2010. [cited at p. 15]

[108] G. Smaragdos, D.A. Khan, I. Sourdis, C. Strydis, A. Malek, and S. Tzilis. A dependable coarse-grain reconfigurable multicore array. In *Parallel Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*, pages 141–150, May 2014. [cited at p. 9]

[109] I. Sourdis, C. Strydis, A. Armato, C.S. Bouganis, B. Falsafi, G.N. Gaydadjiev, S. Isaza, A. Malek, R. Mariani, D.N. Pnevmatikatos, D.K. Pradhan, G.K. Rauwerda, R.M. Seepers, R.A. Shafik, K. Sunesen, D. Theodoropoulos, S. Tzilis, and M. Vavouras. Desyre: On-demand system reliability. *Microprocessors and Microsystems - Embedded Hardware Design*, 37(8-C):981–1001, 2013. [cited at p. 113, 114]

[110] T. Stefanov, C. Zissulescu, A. Turjan, B. Kienhuis, and E. Deprettere. System design using Kahn process networks: the Compaan/Laura approach. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, February 2004. [cited at p. 15]

[111] K. R. Stokke, H. K. Stensland, P. Halvorsen, and C. Griwodz. High-precision power modelling of the tegra k1 variable smp processor architecture. In *2016 IEEE 10th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSOC)*, pages 193–200, Sept 2016. [cited at p. 60]

[112] K. Sudusinghe, S. Won, M. van der Schaar, and S. S. Bhattacharyya. A novel framework for design and implementation of adaptive stream mining systems. In *Proceedings of the IEEE International Conference on Multimedia and Expo*, pages 1–6, 2013. [cited at p. 91]

[113] P. Sundararajan. *High Performance Computing Using FPGAs*. Xilinx, September 2010. [cited at p. 8]

[114] Synopsys®. Design compiler: Rtl synthesis, 2018. https://www.synopsys.com/support/training/rtl-synthesis/design-compiler-rtl-synthesis.html, 2018-08-28. [cited at p. 38]

[115] D. Terpstra, H. Jagode, H. You, and J. Dongarra. Collecting performance data with papi-c. In Matthias S. Müller, Michael M. Resch, Alexander Schulz, and Wolfgang E. Nagel, editors, *Tools for High Performance Computing 2009*, pages 157–173, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. [cited at p. 90]

[116] R. Tessier and W. Burleson. Reconfigurable computing for digital signal processing: A survey. *J. VLSI Signal Process. Syst.*, 28(1-2):7–27, May 2001. [cited at p. 8]

[117] T.J. Todman, G.A. Constantinides, S.J.E. Wilton, O. Mencer, W. Luk, and P.Y.K. Cheung. Reconfigurable computing: architectures and design methods. *IEE Proceedings-Computers and Digital Techniques*, 152(2):193–207, 2005. [cited at p. 8, 10]

[118] N.S. Voros, M. Hübner, J. Becker, M. Kühnle, F. Thomaitiv, A. Grasset, P. Brelet, P. Bonnot, F. Campi, E. Schüler, H. Sahlbach, S. Whitty, R. Ernst, E. Billich, C. Tischendorf, U. Heinkel, F. Ieromnimon, D. Kritharidis, A. Schneider, J. Knaeblein, and W. Putzke-Röming. Morpheus: A heterogeneous dynamically reconfigurable platform for designing highly complex embedded systems. *ACM Trans. Embed. Comput. Syst.*, 12(3):70:1–70:33, April 2013. [cited at p. 114]

[119] D. Wingard. Noc power-management advantages. In *Keynote Talks at IP-SoC Conference and Exibition*, 2013. [cited at p. 38]

[120] Q. Wu, M. Pedram, and X. Wu. Clock-gating and its application to low power design of sequential circuits. *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, 47(3):415–420, March 2000. [cited at p. 38]

[121] Y. Xiao, Y. Xue, S. Nazarian, and P. Bogdan. A load balancing inspired optimization framework for exascale multicore systems: A complex networks approach. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 217–224, Nov 2017. [cited at p. 113]

[122] Xilinx. *Partial Reconfiguration User Guide*, April 2012. [cited at p. 8]

[123] Xilinx. *Vivado Design Suite — AXI Reference Guide — UG1037 (v4.0)*, July 2017. [cited at p. 117, 118]

[124] Xilinx. Ultrascale+ family, 2018. https://www.xilinx.com/about/generation-ahead-16nm.html, 2018-10-7. [cited at p. 8]

[125] H. Xu, R. Vemuri, and W.B. Jone. Run-time active leakage reduction by power gating and reverse body biasing: An energy view. In *26th International Conference on Computer Design, ICCD 2008, 12-15 October 2008, Lake Tahoe, CA, USA, Proceedings*, pages 618–625, 2008. [cited at p. 60]

[126] W. Yang, Z. Li, and Z. Shen. Recognizing and tracking airport runway target in infrared images. In *Proc. National Aerospace and Electronics Conf. NAECON*, 1997. [cited at p. 127]

[127] A.K.W. Yeung and J.M. Rabaey. A reconfigurable data-driven multiprocessor architecture for rapid prototyping of high throughput dsp algorithms. In *System Sciences, 1993, Proceeding of the Twenty-Sixth Hawaii International Conference on*, volume i, pages 169–178 vol.1, January 1993. [cited at p. 10]

[128] J. Yi and J. Kim. Power modeling for digital circuits with clock gating. *IEICE Electronics Express*, 12(24):20150817–20150817, 2015. [cited at p. 60]

[129] F. L. Yuan et al. A multi-granularity fpga with hierarchical interconnects for efficient and flexible mobile computing. *IEEE Journal of Solid-State Circuits*, 50(1):137–149, Jan 2015. [cited at p. 114]

[130] H. Yviquel, A. Lorence, K. Jerbi, G. Cocherel, A. Sanchez, and M. Raulet. Orcc: multimedia development made easy. In *Proceedings of the ACM International Conference on Multimedia*, pages 863–866, 2013. [cited at p. 90]

[131] L. Zhang, J. Yang, C. Xue, Y. Ma, and S. Cao. A two-stage variation-aware task mapping scheme for fault-tolerant multi-core network-on-chips. In *2017 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–4, May 2017. [cited at p. 113]

[132] Y. Zhang, J. Roivainen, and A. Mammela. Clock-gating in fpgas: A novel and comparative evaluation. In *Digital System Design: Architectures, Methods and Tools, 2006. DSD 2006. 9th EUROMICRO Conference on*, pages ‘584–590, 2006. [cited at p. 38]

[133] S. Zhao et al. Sobel-lbp. In *2008 15th IEEE International Conference on Image Processing*, pages 2144–2147, Oct 2008. [cited at p. 127]

[134] D. Zoni and W. Fornaciari. Modeling DVFS and power-gating actuators for cycle-accurate noc-based simulators. *JETC*, 12(3):27, 2015. [cited at p. 60]

List of Publications Related to the Thesis

Published papers

Journal papers

1. Francesca Palumbo, Tiziana FANNI, Carlo Sau, and Paolo Meloni. 2017. *Power-Awarness in Coarse-Grained Reconfigurable Multi-Functional Architectures: a Dataflow Based Strategy.* J. Signal Process. Syst. 87, 1 (April 2017), 81-106.
DOI: https://doi.org/10.1007/s11265-016-1106-9.

2. Francesca Palumbo, Tiziana FANNI, Carlo Sau, Paolo Meloni, and Luigi Raffo, *Modelling and Automated Implementation of Optimal Power Saving Strategies in Coarse-Grained Reconfigurable Architectures.* Journal of Electrical and Computer Engineering, vol. 2016, Article ID 4237350, 27 pages, 2016.
DOI: https://doi.org/10.1155/2016/4237350.

3. Tiziana FANNI, Lin Li, Timo Viitanen, Carlo Sau, Renjie Xie, Francesca Palumbo, Luigi Raffo, Heikki Huttunen, Jarmo Takala, Shuvra S. Bhattacharyya, *Hardware design methodology using lightweight dataflow and its integration with low power techniques.* Journal of Systems Architecture, Volume 78, 2017, Pages 15-29, ISSN 1383-7621.
DOI: https://doi.org/10.1016/j.sysarc.2017.06.003.

4. Lin Li, Carlo Sau, Tiziana FANNI, Jingui Li, Timo Viitanen, Francois Christophec, Francesca Palumbo, Luigi Raffo, Heikki Huttunen, Jarmo Takala, Shuvra S. Bhattacharyya, *An Integrated Hardware/Software Design Methodology for Signal Processing Systems.* Journal of Systems Architecture (2018). DOI: https://doi.org/10.1016/j.sysarc.2018.12.010

Conference papers

1. Tiziana FANNI, Carlo Sau, Luigi Raffo, and Francesca Palumbo. *Automated power gating methodology for dataflow-based reconfigurable systems.* In Proceedings of the 12th ACM International Conference on Computing Frontiers (CF '15), 2015. ACM, New York, Article 61 , 6 pages.
DOI: http://dx.doi.org/10.1145/2742854.2747285

2. Tiziana FANNI, Carlo Sau, Paolo Meloni, Luigi Raffo and Francesca Palumbo, *Power modelling for saving strategies in coarse grained reconfigurable systems.* 2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig), Mexico City, 2015, pp. 1-4.
DOI: 10.1109/ReConFig.2015.7393337

3. Subhadeep Banik, Andrey Bogdanov, Tiziana FANNI, Carlo Sau, Luigi Raffo, Francesca Palumbo, and Francesco Regazzoni. 2016. Adaptable AES implementation with power-gating support. In Proceedings of the ACM International Conference on Computing Frontiers (CF '16). ACM, New York, NY, USA, 331-334.
DOI: https://doi.org/10.1145/2903150.2903488

4. Tiziana FANNI, Carlo Sau, Paolo Meloni, Luigi Raffo, and Francesca Palumbo. 2016. Power and clock gating modelling in coarse grained reconfigurable systems. In Proceedings of the ACM International Conference on Computing Frontiers (CF '16). ACM, New York, NY, USA, 384-391.
DOI: https://doi.org/10.1145/2903150.2911713

5. Tiziana FANNI and Luigi Raffo, *Coarse grain reconfiguration: Power estimation and management flow for hybrid gated systems.* 2016 International Conference on ReConFigurable Computing and FPGAs (ReConFig), Cancun, 2016, pp. 1-4.
DOI: 10.1109/ReConFig.2016.7857160

6. Francesca Palumbo, Carlo Sau, Tiziana FANNI, Paolo Meloni and Luigi Raffo, *Dataflow-Based Design of Coarse-Grained Reconfigurable Platforms.* 2016 IEEE International Workshop on Signal Processing Systems (SiPS), Dallas, TX, 2016, pp. 127-129.
DOI: 10.1109/SiPS.2016.30

7. Lin Li, Tiziana FANNI, Timo Viitanen, Renjie Xie, Francesca Palumbo, Luigi Raffo, Heikki Huttunen, Jarmo Takala, Shuvra S. Bhattacharyya, *Low power design methodology for signal processing systems using lightweight dataflow techniques.* 2016 Conference on Design and Architectures for Signal and Image Processing (DASIP), Rennes, 2016, pp. 82-89.
DOI: 10.1109/DASIP.2016.7853801

8. Carlo Sau, Tiziana Fanni, Paolo Meloni, Luigi Raffo, Maxime Pelcat and Francesca Palumbo, *Demo: Reconfigurable Platform Composer Tool.* 2016 Conference on Design and Architectures for Signal and Image Processing (DASIP), Rennes, 2016, pp. 245-246.
DOI: 10.1109/DASIP.2016.7853835

9. Francesca Palumbo, Carlo Sau, Tiziana FANNI, Luigi Raffo. (2019) *Challenging CPS Trade-off Adaptivity with Coarse-Grained Reconfiguration.* In: De Gloria A. (eds) Applications in Electronics Pervading Industry, Environment and Society. ApplePies 2017. Lecture Notes in Electrical Engineering, vol 512. Springer, Cham. ISBN 978-3-319-93082-4.
DOI: https://doi.org/10.1007/978-3-319-93082-4_8

10. Tiziana FANNI, "Optimal Implementation of Power Saving Techniques in CGR Systems", Cyber-Physical Systems PhD & Postdoc Workshop 2018. Alghero (Italia). CEUR-WS.org/Vol-2208

11. Tiziana FANNI, Alfonso Rodríguez, Carlo Sau, Leonardo Suriano, Francesca Palumbo, Luigi Raffo and Eduardo de la Torre, *Multi-Grain Reconfiguration for Advanced Adaptivity in Cyber-Physical Systems.* 2018 International Conference on ReConFigurable Computing and FPGAs (ReConFig'18). December 2018. Proceedings in press.

12. Alfonso Rodríguez and Tiziana FANNI, *DEMO: Multi-Grain Adaptivity in Cyber-Physical Systems.* Special Session on Energy Efficient Cyber Physical Systems held at the 30th International Conference on Microelectronics (ICM'18). December 2018. Proceedings in press.

Other scientific papers

1. Francesca Palumbo, Carlo Sau, Tiziana FANNI and Luigi Raffo, *Reconfigurable Platform Composer Tool Project*, at the 2016 Riunione Annuale del Gruppo Elettronica (GE), Brescia (Italy) June 2016.

2. Francesca Palumbo, Tiziana FANNI, Carlo Sau, Paolo Meloni and Luigi Raffo, *Automated Flow for Hybrid Clock and Power Gating in Coarse-Grained Reconfigurable Architectures*, at the 2017 Riunione Annuale dell'Associazione SocietÃ Italiana di Elettronica (ex GE), Como (Italy) June 2017.

3. Francesca Palumbo, Rubattu Claudio, Carlo Sau, Tiziana FANNI, Paolo Meloni and Luigi Raffo, *Dynamic Trade-Off Management for CPS*, at the 2017 Italian Workshop on Embedded Systems (IWES), Roma (Italy), September 2017.

4. Carlo Sau, Tiziana FANNI, Luigi Raffo and Francesca Palumbo *Self-adaptation of Cyber-Physical Systems*, at the 2018 Riunione Annuale dell'Associazione SocietÃ Italiana di Elettronica (ex GE), Napoli (Italy) June 2018.

5. Tiziana FANNI, Alfonso Rodríguez, Carlo Sau, Francesca Palumbo, Luigi Raffo and Eduardo de la Torre, *Providing Advanced Adaptivity in Cyber-Physical Systems with Multi-Grain Reconfiguration*, at the 2018 Italian Workshop on Embedded Systems (IWES), Siena (Italy), September 2018.

Appendix A

Logic Regions Algorithms

ALGORITHM 1: Logic Set Definer: baseline Logic Regions identification. ($N = |LR\_MAP|$).

foreach DPN_i *in input* $DPNs$ **do**
  V'_i =mapping of DPN_i in DPN;
  if *isEmpty(LR_MAP)* **then**
    //First iteration
    $S_0 = V'_i$;
    put key S_0 with value DPN_i in LR_MAP;
  else
    foreach S_j *in* LR_MAP *keys* **do**
      if $V'_i = S_j$ **then**
        //V'_i is coincident with an already identified LR
        add DPN_i to value of key S_j in LR_MAP;
        $V'_i = 0$
        break;
      end
      if $V'_i \cap S_j \neq \emptyset$ **then**
        //V'_i partially overlaps with an already identified LR
        $S_N = V'_i \cap S_j$;
        put key S_N with value DPN_i in LR_MAP;
        $S_j = S_j - S_N$;
        $V'_i = V'_i - S_N$;
      end
    end
    if *!isEmpty*(V'_i) **then**
      //The elements left in V'_i need to constitute a new LR
      $S_N = V'_i$;
      put key S_N with value DPN_i in LR_MAP
    end
  end
end

ALGORITHM 2: Logic Set Definer: Logic Regions merging. ($N = |LR\_MAP|$).

while $N >= TH$ **do**
  sort LR_MAP basing on CF;
  S_{lc} =lowest cost S_j in LR_MAP;
  S_{slc} =second lowest cost S_j in LR_MAP;
  put key $S_{mrg} = S_{lc} \cup S_{slc}$ in LR_MAP;
  add value of key S_{lc} to value of key S_{mrg} in LR_MAP;
  add value of key S_{slc} to value of key S_{mrg} in LR_MAP;
  remove keys S_{lc} and S_{slc} from LR_MAP;
end

Appendix B

Logic Regions Identification Algorithm Extension

ALGORITHM 3: Logic Set Definer: Logic Regions set extension with the SBoxes. ($N = |LR\_MAP|$).

foreach SB_i *in* C_TAB **do**
  DPN_{SBi} =new empty set;
  foreach DPN_k *in value of key* SB_i *in* C_TAB **do**
    if *getSBoxValue(*SB_i*,*DPN_k*) !=* X **then**
      add DPN_k to DPN_{SBi};
    end
  end
  add value DPN_{SBi} to key SB_i in SB_MAP
end
foreach SB_i *in* SB_MAP *keys* **do**
  assignedSB=false;
  foreach S_j *in* LR_MAP *keys* **do**
    DPN_{SBi} =value of key SB_i in SB_MAP; $DPNs_j$ =value of key S_j in LR_MAP;
    if DPN_{SBi}=$DPNs_j$ **then**
      add SB_i to key S_j in LR_MAP;
      assignedSB=true;
      break;
    end
  end
  if *!assignedSB* **then**
    $S_N = SB_i$;
    put key S_N with value DPN_i in LR_MAP;
  end
end

Appendix C

Power Analysis Algorithms

ALGORITHM 4: Automatic power saving strategy selection for coarse-grain reconfigurable systems.

```
PG_set is empty;
CG_set is empty;
foreach LR_i in set LRs do
  | evaluate_area(LR_i, area_th)
end
function: evaluate_area(LR_i, area_th):
calculate_LR_i_area;
if area_LR > area_th then
  | evaluate_PG(LR_i);
else
  | evaluate_CG(LR_i);
end
function: evaluate_PG(LR_i):
estimate_PG_total_variation;
if PG_total_variation < 0 then
  | estimate_CG_total_variation;
  | if PG_total_variation < CG_total_variation then
  |   | add LR_i to PG_set;
  | else
  |   | add LR_i to CG_set;
  | end
else
  | evaluate_CG(LR_i);
end
evaluate_CG(LR_i);
function: evaluate_CG(LR_i):
estimate_CG_total_variation;
if CG_total_variation < 0 then
  | add LR_i to CG_set;
end
```

Appendix D

Multi-Grain Adaptivity - Kernel Adaptation Script

Listing D.1: Kernel Adaptation Script.

```
package parser;
import java.util.Scanner;
import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

public class acceleratorParser {
final static String path = "/home/titti/workspace_java/artico3KernelGen/src/parser";
static String input = path + "/inputFile/mm_accelerator.v";
static String output = path + "/outputFile/cgr_accelerator.v";

private static int num_in;
private static int num_out;
private static int num_reg;

private static List<String> signals = new ArrayList<String>();

public static void main(String[] args) {
int i;

acquireData();
acquireSignals();

try{
  BufferedReader inputFile = new BufferedReader(new FileReader(input));
  FileWriter outputFile = new FileWriter(new File (output));

  // Print interface
```

```
    printInterface(outputFile);

    //Print signal declaration
    printSignalsDeclaration(inputFile,outputFile);

    //Copy necessary logic from input file
    copyBody(inputFile, outputFile);

    //Print assignments
    printAssignemnts(outputFile);

    inputFile.close();
    outputFile.close();
  } catch (Exception e) {
    System.out.println("Exception␣found!");
    e.printStackTrace();}

    // replace S01 axi clock and reset signals
    modifyFile(output, "s01_axi_aclk", "clk");
    modifyFile(output, "s01_axi_aresetn", "reset");

    // replace S00 axi clock and reset signals
    modifyFile(output, "s00_axi_aclk", "clk");
    modifyFile(output, "s00_axi_aresetn", "reset");
    modifyFile(output, "clr\\(slv_reg0\\[2\\]", "clr(ready");

    cleanSignals(output);

    // fixing address size and data size
    // TODO: This should be managed in MDC
    modifyFile(output, "SIZE\\(8", "SIZE\\(C_ADDR_WIDTH");

    for(i=0; i < (num_in + num_out); i++){
      modifyFile(output, "slv_reg" + (i+1) + "\\[7", "slv_reg" + (i+1) + "\\[C_ADDR_WIDTH-1");
    }

    modifyFile(output, "\\[7:0\\]␣address_mem_", "\\[C_ADDR_WIDTH-1:0\\]␣address_mem_");
    modifyFile(output, "\\[7:0\\]␣count_", "\\[C_ADDR_WIDTH-1:0\\]␣count_");
    modifyFile(output, "\\[31:0\\]␣data_in_mem_", "\\[C_DATA_WIDTH-1:0\\]␣data_in_mem_");
    modifyFile(output, "\\[31:0\\]␣data_out_mem_", "\\[C_DATA_WIDTH-1:0\\]␣data_out_mem_");
    modifyFile(output, "\\[31␣:␣0\\]␣slv_reg", "\\[C_DATA_WIDTH-1:0\\]␣slv_reg");
  }


  /**
  * Acquire number of input and output ports
  *
  */
  private static void acquireData(){
    Scanner keyboard = new Scanner(System.in);
    System.out.println("Enter␣number␣of␣configuration␣registers");
    num_reg = keyboard.nextInt();

    System.out.println("Enter␣number␣of␣inputs");
    num_in = keyboard.nextInt();
    System.out.println("Enter␣number␣of␣outputs");
```

```
  num_out = keyboard.nextInt();
  keyboard.close();
}

private static void acquireSignals(){
int i;

// add signals not present in modules interfaces
for(i=0; i < (num_in + num_out); i++){
  signals.add("address_mem_"+ (i+1));
  signals.add("data_in_mem_"+ (i+1));
  signals.add("data_out_mem_"+ (i+1));

  if(i < num_in)
    signals.add("wren_mem_"+ (i+1));
  else
    signals.add("rden_mem_"+ (i+1));
  }
  signals.add("done");
  signals.add("done_output");
  signals.add("done_input");

  try{
    String line;
    String tempSignal = "";
    String signal;

    BufferedReader inputFile = new BufferedReader(new FileReader(input));

    while ( (line = inputFile.readLine()) != null){
      if(line.contains("// Coprocessor Front-End(s)")){
        break;
      }
    }

    while ( (line = inputFile.readLine()) != null){
      if(line != null && line.contains(".") &
         line.contains("(") && line.contains(")") && !line.contains(".SIZE")){
        tempSignal = line.split("\\(")[1];
        if(tempSignal.contains("["))
          signal = tempSignal.split("\\[")[0];
        else
          signal = tempSignal.split("\\)")[0];

        if(!signals.contains(signal))
          signals.add(signal);
      }
    }
    inputFile.close();
  } catch (Exception e) {
  System.out.println("Exception found!");
  e.printStackTrace();}

  if(signals.contains("start"))
  signals.remove("start");
}
```

```
static void printInterface(FileWriter outputFile){
int i;
// Printing interface

  try {
  outputFile.write("//------------------------\n");
  outputFile.write("// Module interface \n");
  outputFile.write("//------------------------\n");

  outputFile.write("module cgr_accelerator#(\n"+
  "parameter integer C_DATA_WIDTH = 32, // Data bus width\n" +
  "parameter integer C_ADDR_WIDTH = 16 // Address bus width\n\n)\n(" +
  "// Global signals\n" +
  "input clk,\n" +
  "input reset,\n\n" +
  "// Control Signals\n"+
  "input wire start,\n" +
  "output reg ready,\n\n");

  outputFile.write("// Configuration registers \n");
  for (i= 0; i<num_reg; i++){
    outputFile.write("output wire [C_DATA_WIDTH-1 : 0] reg_" + i + "_o,\n" +
    "output wire reg_" + i + "_o_vld,\n" +
    "input [C_DATA_WIDTH-1 : 0] reg_" + i +"_i,\n\n");
  }

  outputFile.write("    // Data memories \n");
  for(i=0; i< (num_in + num_out); i++){
    outputFile.write("  output bram_"+ i +"_clk,\n" +
    "output bram_"+ i +"_rst,\n" +
    "output wire bram_" + i + "_en,\n" +
    "output wire bram_" + i + "_we,\n" +
    "output wire [C_ADDR_WIDTH-1 : 0] bram_" + i + "_addr,\n" +
    "output wire [C_DATA_WIDTH-1 : 0] bram_" + i + "_din,\n" +
    "input wire [C_DATA_WIDTH-1 : 0] bram_" + i + "_dout,\n\n");
  }

  outputFile.write("    // Data Counter \n" +
  "input [31 : 0] values);\n\n\n");
  // Ending printing interface

  } catch (IOException e) {
  e.printStackTrace();}
}

static void printSignalsDeclaration(BufferedReader inputFile, FileWriter outputFile){
String line;
int i;

  try {
  while ( (line = inputFile.readLine()) != null){
    if(line.contains("// Module Signals"))
      outputFile.write("// ------------------------\n");
    outputFile.write(line + "\n");
```

```
      break;
  }

  while ( (line = inputFile.readLine()) != null){
    if(!line.contains("// Body"))
      outputFile.write(line  + "\n");
    else
      break;
  }
  outputFile.write("// End module signals declaration\n");
  } catch (IOException e) {
  e.printStackTrace();}
}

static void copyBody(BufferedReader inputFile, FileWriter outputFile){

//Copy necessary logic from input to output file
String line;
  try {
    while ( (line = inputFile.readLine()) != null){
      if(line.contains("// Coprocessor Front-End(s)"))
        break;
    }

    outputFile.write("// Logic to manage ready signal\n"
    + "// -------------------------\n"
    + "always@(posedge clk or negedge reset)\n"
    + "if(!reset)\n"
    + "ready <= 1;\n"
    + "else\n"
    + "if(start) ready <= 0;\n"
    + "else if(done) ready <= 1;\n\n"
    + "\n\n// -------------------------\n"
    + "// Coprocessor Front-End(s)\n");

    while ( (line = inputFile.readLine()) != null){
      if(!line.contains("endmodule"))
        outputFile.write(line  + "\n");
      else
        break;
    }
  } catch (IOException e) {
  e.printStackTrace();}
}

static void printAssignemnts(FileWriter outputFile){
int i;

  try {
    //assignments
    for(i=0; i< (num_in + num_out); i++){

    if(i<num_in){
      outputFile.write("\nassign bram_" + i + "_rst = !reset;\n" +
      "assign bram_" + i + "_en = rden_mem_" + (i+1) + ";\n" +
      "assign bram_" + i + "_we = wren_mem_" + (i+1) + ";\n" +
```

```
      "assign␣bram_" + i + "_addr␣=␣address_mem_" + (i+1) + ";\n" +
      "assign␣bram_" + i + "_din␣=␣data_in_mem_" + (i+1) + ";\n" +
      "assign␣data_out_mem_" + (i+1) + "␣=␣bram_" + i + "_dout;\n");
    }
    else{
      outputFile.write("\nassign␣bram_" + i + "_rst␣=␣!reset;\n" +
      "assign␣bram_" + i + "_en␣=␣wren_mem_" + (i+1) + ";\n" +
      "assign␣bram_" + i + "_we␣=␣wren_mem_" + (i+1) + ";\n" +
      "assign␣bram_" + i + "_addr␣=␣address_mem_" + (i+1) + ";\n" +
      "assign␣bram_" + i + "_din␣=␣data_in_mem_" + (i+1) + ";\n" +
      "assign␣data_out_mem_" + (i+1) + "␣=␣bram_" + i + "_dout;\n");
    }
    }

    for(i=0; i< (num_reg); i++){
      outputFile.write("\nassign␣slv_reg" + i + "␣=␣reg_" + i + "_i;\n" +
      "assign␣reg_" + i + "_o_vld␣=␣1'b0;\n" +
      "assign␣reg_" + i + "_o␣=␣{C_DATA_WIDTH{1'b0}};\n");
    }

    outputFile.write("\nendmodule\n");
  } catch (IOException e) {
  e.printStackTrace();}
}

static void modifyFile(String filePath, String oldString, String newString){

File fileToBeModified = new File(filePath);
String oldContent = "";
BufferedReader reader = null;
FileWriter writer = null;

  try
  {
    reader = new BufferedReader(new FileReader(fileToBeModified));

    //Reading all the lines of input text file into oldContent
    String line = reader.readLine();
    while (line != null) {
      oldContent = oldContent + line + System.lineSeparator();
      line = reader.readLine();
    }

    //Replacing oldString with newString in the oldContent
    String newContent = oldContent.replaceAll(oldString, newString);

    //Rewriting the input text file with newContent
    writer = new FileWriter(fileToBeModified);
    writer.write(newContent);
  }
  catch (IOException e){
  e.printStackTrace();}

  finally       {
    try {
      //Closing the resources
```

```
      reader.close();
      writer.close();
      }
    catch (IOException e) {
    e.printStackTrace();}
  }
}

private static Boolean copyLine(String line) {
Boolean copy = true;
String signal = "";

  if (line.contains("wire␣") || line.contains("reg␣")){
    if (line.contains("wire␣") ){
      signal = line.split("wire␣")[1];
      signal = signal.split(";")[0];
      if (signal.contains("]")) {
        signal = signal.split("\\]")[1];
    }
  }

  if (line.contains("reg␣") ){
    signal = line.split("reg␣")[1];
    signal = signal.split(";")[0];
    if (signal.contains("]")) {
      signal = signal.split("\\]")[1];
    }
  }

  if (signal.contains("␣")){
    signal = signal.split("[\\s\\xA0]+")[1];
  }

  if(signals.contains(signal))
    copy = true;
  else
    copy = false;
  }
  return copy;
}

static void cleanSignals(String filePath){

File fileToBeModified = new File(filePath);
String newContent = "";
BufferedReader reader = null;
FileWriter writer = null;
String line;

  try {
    reader = new BufferedReader(new FileReader(fileToBeModified));

    //Copy  all the lines of input text file until signals declaration starts
    while ((line = reader.readLine()) != null) {
      if(line.contains("//␣Module␣Signals"))
        break;
```

```
      else
        newContent = newContent + line + System.lineSeparator();
    }

    // copy only necessary signals, from signals declaration
    while ((line = reader.readLine()) != null) {
      if(line.contains("// End module signals declaration"))
        break;
      else
        if(copyLine(line))
          newContent = newContent + line + System.lineSeparator();
    }

    //Copy  all the lines of input text file until the end of the file
    while ((line = reader.readLine()) != null) {
      newContent = newContent + line + System.lineSeparator();
    }

    //Rewriting the input text file with newContent
    writer = new FileWriter(fileToBeModified);
    writer.write(newContent);

    reader.close();
    writer.close();
  }
  catch (IOException e){
  e.printStackTrace();}
}
}
```