Automated Generation of Heterogeneous Multiprocessor Architectures: Software and Hardware Aspects

Youenn Corre

THÈSE / UNIVERSITÉ DE BRETAGNE SUD
sous le sceau de l'Université Européenne de Bretagne

Pour obtenir le grade de :
DOCTEUR DE L'UNIVERSITÉ DE BRETAGNE SUD
Mention : STIC
École Doctorale SICMA

présentée par

Youenn Corre

Laboratoire des Sciences et Techniques de l'Information, de la Communication et de la Connaissance

Automated Generation of Heterogeneous Multiprocessor Architectures: Software and Hardware Aspects

Thèse soutenue le 23 janvier 2013,
devant la commission d'examen composée de :

Pr. Smail Niar
Professeur, LAMIH - Université de Valenciennes / Président

Pr. Daniel Ménard
Professeur, INSA Rennes / Rapporteur

Pr. Frédéric Rousseau
Professeur, TIMA - UJF Grenoble / Rapporteur

Dr. Dominique Heller
Ingénieur de Recherche, Lab-STICC - UBS Lorient / Encadrant de thèse

Pr. Loïc Lagadec
Professeur, Lab-STICC - ENSTA Bretagne / Co-directeur de thèse

Dr. Jean-Philippe Diguet
Directeur de Recherche CNRS, Lab-STICC - UBS Lorient / Directeur de thèse

Résumé

Les systèmes embarqués sont aujourd'hui omniprésents et les progrès d'intégration accompagnant cette évolution permettent d'accroître leurs fonctionnalités et capacités potentielles. Cette co-évolution a conduit à l'emergence des systèmes-sur-puce multiprocesseurs hétérogènes qui répondent aux contraintes des systèmes embarqués en termes de performances et d'énergie. Cependant cet avantage se traduit par une complexité de conception et de programmation accrue. Le niveau d'expertise requis ainsi que le temps de développement limitent considérablement leur déploiement, il est donc nécessaire de réaliser des outils permettant d'affranchir les concepteurs des détails architecturaux et de programmation afin qu'ils puissent mobiliser leurs efforts sur les étapes à forte valeur ajoutée. L'objectif est donc d'automatiser les tâches fastidieuses et chronophages propres à la conception d'architectures multiprocesseurs hétérogènes, notamment sur FPGA, en élevant le niveau d'abstraction selon une approche qui unifie la synthèse de haut-niveau et la co-conception logicielle/matérielle au-delà des approches existantes qui se révèlent partielles ou inadaptées.
Les travaux de cette thèse sont une réponse à ce problème, ils présentent un outil de conception reposant sur le principe d'une automatisation des tâches fastidieuses et laissant la main au concepteur là où celui-ci le souhaite. Pour cela, on s'appuie sur un modèle d'architecture défini à l'aide d'un formalisme de haut-niveau indépendant des détails d'implémentation, palliant ainsi l'absence d'architecture multiprocesseur sous-jacente dans les FPGA. Ce modèle de spécification permet également au concepteur de fournir les contraintes à différents niveaux de détails en fonction de ses connaissances du système ou de son niveau d'implication. L'exploration de l'espace de conception se fait grâce à un algorithme *scalable* et reposant sur des estimations rapides et précises. Une méthode d'exploration des accélérateurs matériels, utilisant la synthèse de haut-niveau pour une estimation rapide des coûts, est introduite. Enfin, l'intégration de méthodes d'ingénierie dirigée par les modèles permet la génération du design final et notamment des fichiers d'implémentation en fonction de la cible, facilitant ainsi la portabilité et la réutilisation des designs. L'outil a été validé à travers deux études de cas : un décodeur vidéo MJPEG et une application complexe de détection de visage.

Mots-clés: Systèmes-sur-puce multiprocesseurs hétérogènes, FPGA, Conception électronique assistée par ordinateur.

Abstract

Embedded systems are now ubiquitous and the increase in the integration capacity allows for more features and capabilities. This trend has led to the emergence of Heterogeneous Multiprocessors Systems-on-Chip (H-MPSoC) which provide a way to respect the cost and performance constraints inherent to embedded systems. However they also make the task of designing and programming such systems a long and arduous process. The skills required along with the long development time are obstacles to their diffusion. It is thus necessary to develop tools that will free designers from architectural and programming details, so that they can focus on the tasks where they can bring added-value. The objective is thus to automatize the tedious tasks that burden the design of H-MPSoC, in particular on FPGA, by providing a higher-level of abstraction following a method that brings together High-Level Synthesis and hardware/software co-design beyond the existing solutions which are whether incomplete or unfit.
The presented work aims at providing an answer to these problems. They introduce a design framework relying on the automation of tedious tasks and allowing designers to express their expertise where they want to. For this, we rely on an architecture model defined with a high-level formalism independent from implementation details, providing a solution to the lack of multiprocessor architecture in FPGAs. This specification model also allows designers to provide design constraints in accordance with their level of expertise or involvement. The design space exploration is implemented as a scalable algorithm relying on fast and accurate estimation techniques. A method for the exploration of hardware accelerators based on high-level synthesis to provide fast cost estimations is introduced. Finally the integration of model-driven engineering methods enables portability and reuse by generating the final design implementation. The framework is validated through two case studies: an MJPEG video decoder and a more complex face detection application.

Keywords: Heterogeneous Multiprocessors Systems-on-Chip, FPGA, Electronic-System Level Design.

n° d'ordre :

Université de Bretagne-Sud

Laboratoire Lab-STICC - CNRS, UMR 6285
Centre de Recherche Christiaan Huygens - rue de Saint-Maudé - BP 92116 - F-56321 Lorient Cedex - FRANCE
Tél : + 33(0)2 97 87 46 20 Fax : + 33(0)2 97 87 45 27

Atchoum!

Victor Hugo, un jour de grand froid.

Remerciements

Je remercie mon Directeur de thèse, Jean-Philippe Diguet, pour son encadrement, sa grande disponibilité, ses nombreuses idées et ses conseils tout au long de la thèse. Merci à Loïc Lagadec, qui a été mon encadrant principal durant la première année, pour ses remarques parfois difficiles à entendre mais toujours justes et pertinentes. Je remercie Dominique Heller pour son aide et son expertise qui m'ont été très utiles durant ces trois ans. Merci également à Koen Bertels de m'avoir accueilli pour trois mois au sein de son équipe à TU Delft.

Je remercie M. Smail Niar d'avoir accepter de présider mon jury de thèse. Je remercie également MM. Daniel Ménard et Frédéric Rousseau d'avoir réaliser les rapports du présent mémoire malgré le peu de temps dont ils disposaient.

Je remercie les nombreuses personnes que j'ai eu l'opportunité de croiser au cours de ces trois ans : l'équipe Architectures & Systèmes de Brest, l'équipe de *Computer Engineering* à l'Université technologique de Delft et les membres du Lab-STICC Lorient pour l'ambiance unique qui y règne. Merci également aux nombreux doctorants que j'ai rencontrés, trop nombreux pour être tous cités ici, et avec qui j'ai passé de bons moments.

Enfin, je remercie mes amis et ma famille, mon père, ma mère et Brenda, ma grande sœur, pour leur soutien tout au long de cette thèse.

Contents

1 Introduction

Embedded systems are now more than ever part of daily life. This ubiquity also means diversity, as embedded systems can take many forms, from a portable camera to system control in airplanes and include smartphones, set top boxes, cameras, GPS, etc. All these apparels tend to grow in features and consequently their designs also grow in complexity.

In order to be able to deal with the typical constraints of embedded systems — performance, size and power consumption — in spite of the increasing complexity, a new type of system was developed: System-on-Chip (SoC). SoCs gather on one single chip all the necessary components for a complete system, i.e. processing units, memories, communication buses, peripheral controllers, Analog/Digital converters and so on. SoCs are thus an efficient way to provide the required performances while reducing the size and the power consumption of the system. Designing SoCs however is a complex task that requires skill and knowledge in order to balance performances, size and power consumption. When designing such systems, lots of decisions have to be taken due to the large number of design options, leading to larger design space. Due to the increasing complexity of SoCs, such decisions can no longer be taken manually without being a highly error-prone process and cannot be performed within a reasonable time.

Heterogeneity SoCs have then evolved toward multiprocessor architectures, to answer the growing demand for computation power, leading to homogeneous Multiprocessor Systems-on-Chip. This introduced parallel programming at the software level in the design flow, along with the problems inherent to this programming paradigm, i.e. mapping, communication and synchronization. Then MPSoCs evolved toward Heterogeneous Multiprocessor Systems-on-Chip, which include specialized processing units, in order to answer the necessity to have better system performance and energy-efficiency. This has also brought new design difficulties: partitioning and design of hardware accelerators. These new design steps were out of scope of software engineers and thus required new skills. This thus lead to the necessity for companies to adapt by hiring hardware engineers, thus leading to increased development costs making development of such systems riskier, especially for Small and Medium Enterprises (SME).

Off-the-shelf H-MPSoCs are available, for instance the Texas Instruments OMAP processors [1] are a family of H-MPSoC present in many smartphones. However these

H-MPSoCs have fixed architectures and are usually designed for widespread application domains. Consequently, they cannot be easily adapted to specific domain and can thus be oversized in regards to the system requirements. Moreover, the specialized processing units cannot be easily programmed.

H-MPSoC is a domain where innovation is still active. In 2012, Xilinx has released the Zynq Field-Programmable Gate Array (FPGA) board [2], which associates a multiprocessor SoC with FPGA capabilities, bringing reconfigurability into SoCs. The advantage means that it is possible to adapt the hardware accelerators accordingly with the current needs of the system. Another advantage is the possibility to adapt the system to evolution, which is useful especially in long-life system such as military applications, satellites, cyber-physical systems, etc.

FPGA Design Complexity The development of systems such as SoCs usually goes through a prototype phase, prototype which is typically implemented on a FPGA for validation. However designing for FPGA is still a costly process, despite the design tools provided by FPGA manufacturers. In [3], a comparison is made in the development of a high-performance application on different targets: an x86-based General Purpose Processor (GPP), a Graphical Processing Unit (GPU) and an FPGA. The study clearly shows the advantages of FPGA in terms of energy-efficiency and performances over the other solutions. For the studied application, the speedup over GPP is of two orders of magnitude with an energy consumption three orders of magnitude lower than the GPP. However the study also shows that it remains a very long and very costly process, most of the cost coming from the long development time. The development time is two orders of magnitude greater than the one for GPP with a cost one order of magnitude greater. The authors of the study suggest that the reasons of these flaws in FPGA design are due to the difficulty to master FPGA design and debug processes. They also point out the lack of standards which prevents all backward and onward compatibility, thus prohibiting design portability and reuse.

Moreover FPGA design is a very time-consuming process due to the different stages of the implementation: synthesis, mapping, place and route and test. This fact supports the point of view in the industry that FPGA is a risky product, which has low productivity and is consequently associated with longer time-to-market. Hence the necessity to have a way to reduce the implementation time by providing accurate pre-implementation evaluations of a design both in size and performance and to provide validated and thus bug-free systems in order to minimize the debug phase.

Design Tools Tediousness Along with chip capabilities, tools complexity has increased from time to time and another difficulty encountered by designers is thus the tediousness of the design tools that contain numerous configuration options which can be confusing and can easily lead to make mistakes consequently very hard to debug. For these reasons, it is necessary to find tools to simplify the design process of such systems and thus increase the designer productivity. This can be done by automating the tedious parts of the design process in which designers cannot bring any added-value.

In embedded systems, the frontier between hardware and software domains is fuzzy and consequently the design of one is dependent of the other. This is problematic since typical software engineers and hardware engineers have design approaches that differ greatly and they are usually not aware on how things are done in the other domain. There are few engineers who master how both worlds work and thus it is a rare and valuable skill. That problem can be overcome by providing on the one hand a high-level of abstraction in which the distinction between hardware and software is not made, and on the other hand by automating the design decisions and in particular the partitioning between hardware and software implementations.

1.1 Contributions

All these reasons show that the design of H-MPSoC for FPGA is a tedious, error-prone and costly process, leading to the necessity to have a tool to assist designers in order to simplify and speedup the design process. In this thesis, we present a framework for the automation of the design of efficient H-MPSoC. Given an application, a basic architecture template and a set of constraints, the framework explores the design space and generates a complete system including the synthesizable hardware platform and the adapted C code respecting the provided constraints. To achieve this goal, several dimensions are explored: the number and type of processors, the hardware specialization, the data parallelism exploitation, the communication and memory models, mappings and schedulings. However designers have the control of the flow and can introduce design decisions according to their design skills.

The contributions of this thesis are:

- A framework that automates the tedious phases of the H-MPSoC design in order to let designers focus where their knowledge is most needed. Our framework also generates a correct-by-design code for both the hardware and software implementations. This avoids the error-prone design process provided by current FPGA design tools, and since it lowers the cost of the design process to the implementation, more designs can be further explored.

- An automated and scalable Design Space Exploration (DSE) algorithm that takes into account size and performance constraints and relies on fast and accurate estimators. This simplifies the design by exploring the numerous design options of H-MPSoCs and by evaluating them in order to propose a set of Pareto-optimal options to the designer. Explored dimensions include hardware accelerator choice, processor allocation, task mapping, memory selection and data mapping. The scalability factor also allows the designer to balance between the time spent for optimizing the results, consequently optimizing the designer productivity.

- The integration of a fast and automated exploration of hardware accelerators in the DSE loop by means of High-Level Synthesis (HLS). This offers more design options by providing a tradeoff between cost and performances and allows to fully exploit the heterogeneity of H-MPSoC.

- The integration in the DSE loop of the data parallelism exploration through task duplication in order to further speedup the designed system.
- The use of Model-Driven Engineering (MDE) methods through the integration of a template-based approach to describe the input architecture. This template let designers configure the exploration according to their own level of expertise, while also abstracting away target-specific implementation details. This also favors design portability and reuse by providing a solution to the absence of FPGA standards.
- A strategy favoring reuse through the use of databases in order to speedup the design process.

1.2 Outline

This manuscript is structured as follows:

- Chapter 2 describes the state of the art in ESL design tools for MPSoC, in HLS tools, and in the usage of MDE for DSE of MPSoC.
- Chapter 3 presents an overview of the framework flow.
- Chapter 4 introduces our DSE algorithm: the explorations methods are detailed along with the used estimation techniques for cost and performance evaluation.
- Chapter 5 explains how we use MDE methods to simplify, to favor reuse and to make more reliable the design of H-MPSoC
- Chapter 6 presents the evaluation and validation of our framework through the use of two case-studies and several benchmarks.
- Chapter 7 concludes this thesis by summarizing its contributions and providing perspectives on future work.

2
State of the Art

In this chapter we present the current state of the research in ESL tools for MPSoC, as well as for more specific domain related to a couple of contributions of our framework: the integration of HLS and the usage of Model-Driven Engineering approach in MPSoC design.

2.1 Ideal ESL Framework

In this section, we describe what we think would be the ideal framework flow, without caring about the feasibility aspect of it.

Adaptability

This ideal framework should be adapted to several levels of expertise of designers: from software designers who have only little knowledge in hardware design to expert system designers. The former want to deal as little as possible with the hardware aspects and thus the framework should hide this level, should take all the design decisions and provides at the end a ready-to-implement system that satisfies the designer's constraints. Expert designers on the other hand, would use such framework in order to be relieved from tedious tasks that can be automated such as design cost and performance evaluation, software adaptation, communication interfaces generation, etc. This means that advanced-level designers should be able to express their expertise by being able to enforce design decisions they know will yield the best results.

Inputs

So this ideal framework must allow designers to express their wishes with minimal efforts. On the application side, that would mean to be able to use the application code as input without any modifications and no limitation on the expressibility of the coding language accepted by the tool which, ideally, should be a widespread language (C, C++, etc.) or model formalism (UML, etc.). However that constraint is not realistic since most frameworks constrain only accept applications expressed in one or more Models of Computation (MoC). The architecture specifications/constraints should also be easy to express: this can be done through the use of component library and usage of a Graphical User Interface (GUI) that would help the designer visualize the design possibilities given

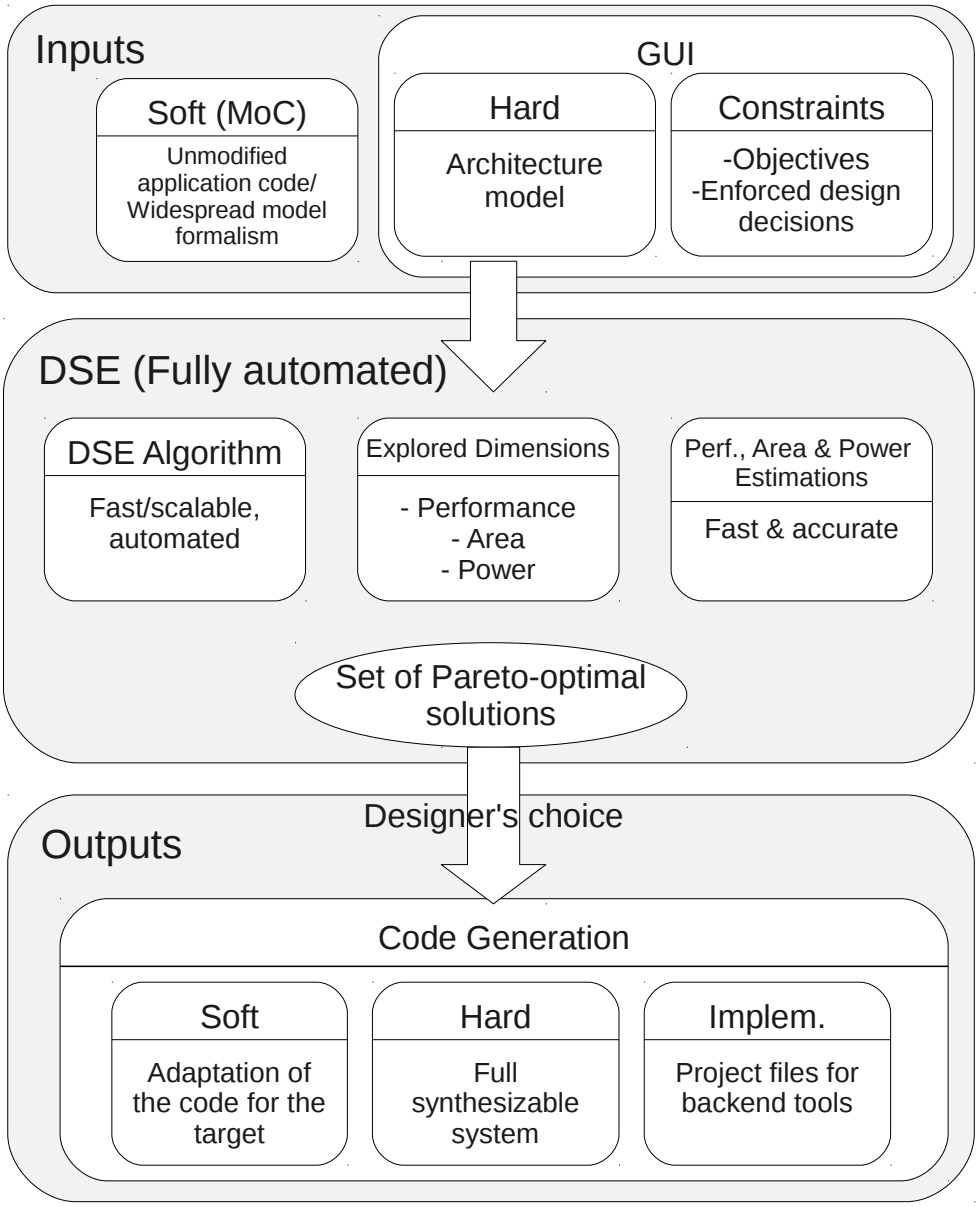

Figure 2.1: Ideal design flow for an H-MPSoC.

the current specifications and constraints. The tool should also give the possibility to advanced designers to express their expertise by letting them enforce specific design decisions. The characterization of the inputs should also be left as little as possible to the designers: for instance the software execution time of a task on a specific target should be measured through an automated on-target profiling or through an Instruction Set Simulator (ISS), the logic resource cost of hardware components should be provided whether from manufacturers documentation or through automated methods based on behavioral or logic synthesis, and so on.

Design Space Exploration

Once the inputs are specified, the rest of the process should be fast, entirely automated and should return a ready-to-implement complete system that meets the designer's goal. Thus the DSE process:

- should be able to explore as many dimensions as possible: performance, area, power, etc. in order to offer designers a maximum of options;
- should be fast to explore a large number of designs or at least be scalable to let the designer choose between DSE speed and performance of the results;
- should integrate available debug and test tools;
- and should provide accurate estimations that faithfully reflect the characteristics of the final design.

Outputs

The results of the DSE should be a set of Pareto optimal solutions giving a large number of quality solutions that offer tradeoffs between several criteria to the designer. The implementation files for the selected design should then be generated: the software code of the application is adapted, the hardware platform is synthesized and project files for implementation are generated, so that it can be directly implemented using the backend tool corresponding to the selected target (FPGA, ASIC, etc.).

2.2 Existing MPSoC Design Tools

Over the years, numerous tools and frameworks have been built for the design of architecture at the Electronic System-Level (ESL). This section presents a selection of such tools, and states the differences between them and our framework.

2.2.1 Daedalus

Daedalus [4] is a framework for the design of Heterogeneous MPSoC developed at the university of Amsterdam and Leiden. It is made of several tools, each performing a

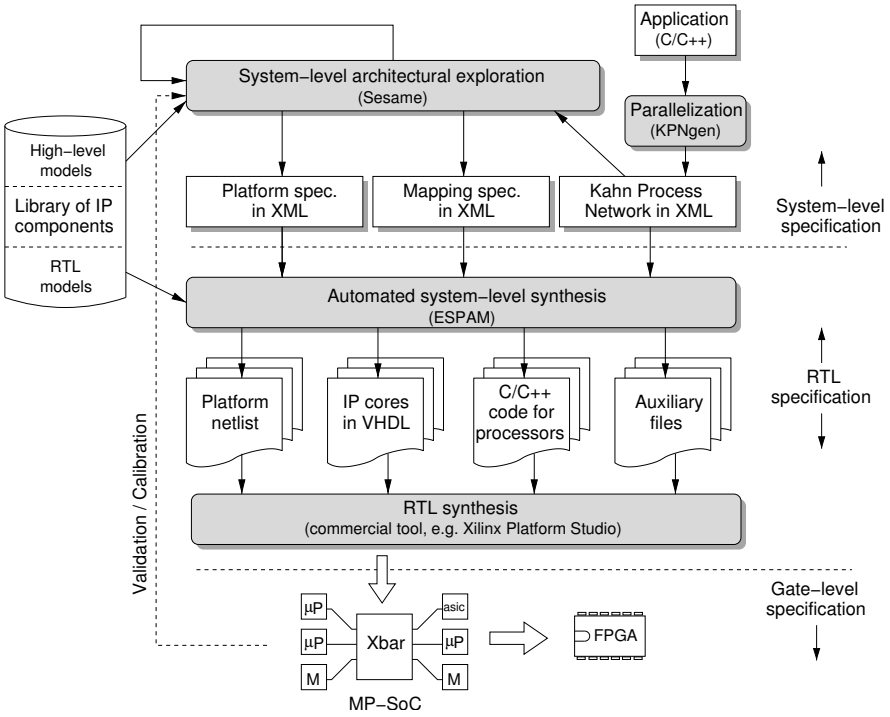

Figure 2.2: Design flow of Daedalus. Figure coming from [4].

different part of the design. KPNGen [5] generates a parallel version of the input application under the Kahn Process Network (KPN) formalism. Espam [6] deals with the code generation and the implementation files. Sesame [7] is a trace-based simulator which provides communication and performance estimations in order to evaluate the design under exploration. Daedalus starts by transforming the input sequential application into a Kahn Process Network version in order to exploit task-parallelism. For this it uses a tool named KPNGen that, provided that the input fulfill the requirements, automatically performs a series of transformations on the application: first it is transformed into a Single Assignment Code and from that version it produces the Polyhedral Reduced Dependency Graph (PRDG), and then the final KPN form of the application.

Then the KPN description is used as input for the Design Space Exploration. The explored dimensions are the number and the types of processing units, mappings and scheduling. The evaluation is based on traces of the application events during execution, which are read, write and execute events. These events are then used to simulate the execution of the application on the currently evaluated architecture. The results is a set of statistics which will help the designer chose the system best-fitting to his needs. Then the ESPAM tool checks some design rules constraints and generates the corresponding synthesizable code for implementation of the corresponding MPSoC solution.

Daedalus' design space exploration is based on an exhaustive exploration of the solutions based on the constraints given by the designer as input. Such an approach

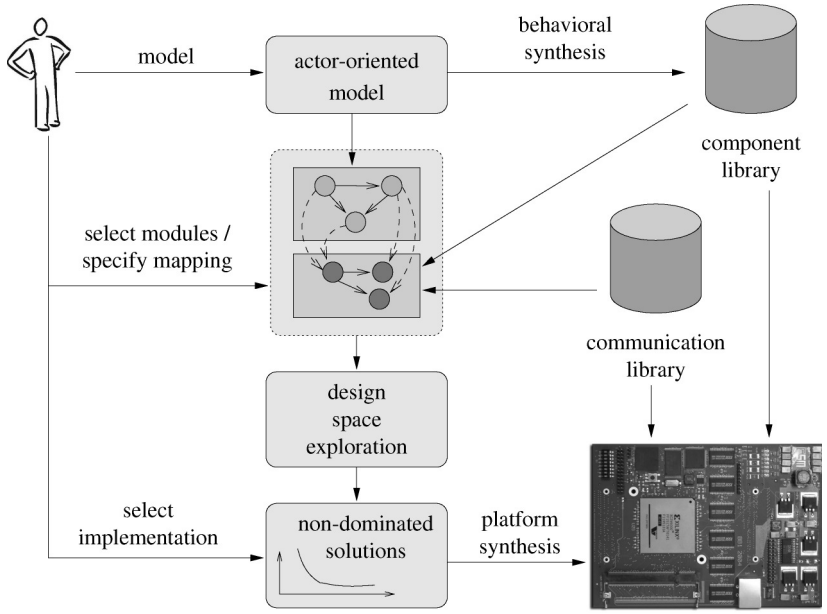

Figure 2.3: Design flow of SystemCoDesigner. Figure coming from [8].

can be terribly slow for loose exploration constraints leading to design spaces which cannot be entirely explored within a reasonable time.

2.2.2 SystemCoDesigner

SystemCoDesigner [8] is an automated framework for the exploration and the generation of the hardware and software of System-on-Chip for FPGA and developed at the university of Erlangen-Nuremberg. It uses as input an application described as an actor-oriented model communicating through FIFO in SysteMoC [9], a subset of the SystemC language. Then each actor undergo a behavioral synthesis with Forte Cynthesizer [10] in order to produce the corresponding hardware accelerator. In order to get accurate software execution times of the application, which are used for performance estimation, they have implemented a non-intrusive on-target profiling based on a hardware timer.

The design space exploration then begins, using a multiobjective evolutionary algorithm (MOEA). During DSE, partitioning and mapping decisions are taken for each actor and then performance of the designs, given as latency and throughput, are accurately estimated using Virtual Processing Components (VPC), an event-based simulator. The area cost is estimated based on the cost of the hardware components. Implementation of the communication are decided at this stage depending on the type of communication (software to hardware or hardware to hardware) and the available kinds of memory (e.g. BRAM, LUT, etc.). Since MOEA is a non-deterministic algorithm, several iterations can be performed in order to maximize the resulting set of non-dominated solutions. The final selected design is then generated for implementation

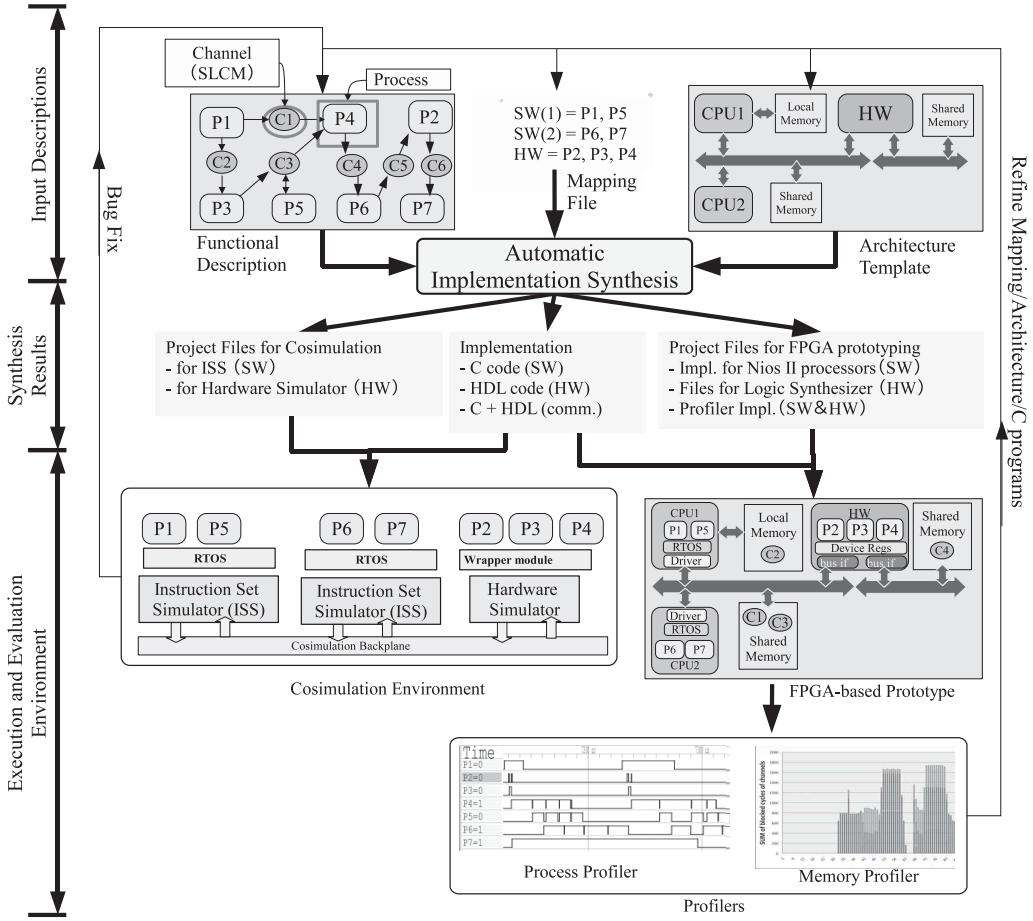

Figure 2.4: Design flow of Advanced Systembuilder. Figure coming from [11].

on the target FPGA: processors are instantiated as softcores, hardware accelerators are added to the design, communication interfaces are derived from the application model and are provided as a RTL description, the SysteMoC models of actors implemented in software are automatically converted to C++ and finally a bitstream for the target FPGA is generated.

In [8] are presented the results of an exploration with SystemCoDesigner for an MJPEG decoder. Although scalable, the DSE took over 2 days and 17 hours to evaluate 7600 solutions resulting in a set of 366 non-dominated solutions which is relatively slow compared to other solutions. They integrate HLS in their design flow however they do not take advantage of it to explore the possible tradeoffs between cost and performance by producing a series of accelerators with different characteristics, thus limiting their design space.

2.2.3 Advanced Systembuilder

Advanced Systembuilder [11] is a framework for automated synthesis of H-MPSoCs developed by the universities of Nagoya and Ritsumeikan. The inputs of the tool are:

- a functional description of the system using processes and channels with *System Level Communication Model* (SLCM);
- an architecture template which specifies the architecture with the available hardware components (number of processors, number of hardware accelerators, and the number of types of memories) and how they are assembled;
- a mapping specification.

These inputs must be manually specified by the designer. From these inputs, Advanced Systembuilder automatically perform several synthesizes: the hardware is generated through behavioral synthesis with the eXCite tool; the software is synthesized as a C code targeting a Real-Time Operating System (RTOS); communication are synthesized as Application Programming Interface (API) in the C code and as hardware components. In addition, two implementation outputs are produced: one is the files for cosimulation and the other one is the implementation files for typical FPGA backend tools such as Altera Quartus or xilinx ISE. These implementations are used to validate and evaluate the performance of the design.

To evaluate the performance of the prototype, Advanced Systembuilder seamlessly integrates into the FPGA design non-intrusive hardware memory and process profilers. The former measures the accesses to the memory along with the potential contention that can occur on the bus accessing the memory, and the latter profile the execution and idle time of the processes. With these measurements designers can decide if they are satisfied with the current design or can identify the points that needs improvement in his design (bottlenecks) and thus modify the design accordingly and relaunch a new iteration of Advanced Systembuilder with it.

The biggest lack in Advanced Systembuilder is that it does not perform any automated Design Space Exploration since mapping specifications and the hardware description of the system are provided by the designer as inputs. Moreover to evaluate the performance of a design, they whether rely on a slow Cycle-Accurate Byte-Accurate (CABA)simulation with ModelSim or onto an implementation on FPGA which implies a long implementation time. Consequently, exploring several designs is relatively slow: in [11], the evaluation of 24 designs required about five hours, most of it being due to the logic synthesis steps.

2.2.4 hArtes

hArtes [12] (*Holistic Approach to Reconfigurable Real-Time Embedded Systems*) is an European project which aims at developing a framework for hardware/software codesign. Three input formalisms are accepted for the application specification: C code, Scilab description (a software for numerical computation equivalent to MATLAB) or

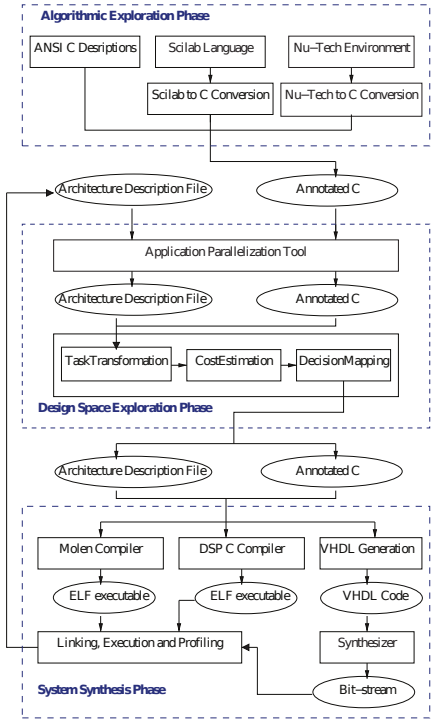

Figure 2.5: Design flow of hArtes. Figure coming from [12].

Nu-Tech description, a tool to design application in a graphical way. If the input is one of the latter two solutions, the equivalent C code is generated from the specifications in order to be used as input for the DSE. This C code is annotated with three different kinds of pragmas: pragmas that specify the possible parallelism of a section of code, pragmas that specify the profiling information of the program functions and pragmas that indicate tasks that can be accelerated through hardware. Those pragmas can be inserted whether manually by the designer or automatically by the tool during DSE. The hardware architecture specification is given as an XML file.

hArtes targets MOLEN architecture [13], which consists in the association of GPPs and specialized processing units (Intellectual Property (IP), Digital Signal Processor (DSP), etc.) on a reconfigurable unit. The application first starts by being parallelized with a dedicated tool. Then the DSE performs the mapping of the tasks, and the designs are then evaluated using the SoCLib modeling and simulation tool [14]. Then begin the so-called System Synthesis phase during which the annotated C code is compiled according to the mapping decision: whether with the MOLEN Compiler or with a DSP compiler. If a task is implemented in hardware, then the Dwarv tool [15] is called and generates an HDL version of the accelerator from the C code. So the final system implementation is the executable of the application and the bitstream of the hardware architecture.

The fact that hArtes allows several formalisms for the application specification allows to target a larger audience who may not be familiar with classical program-

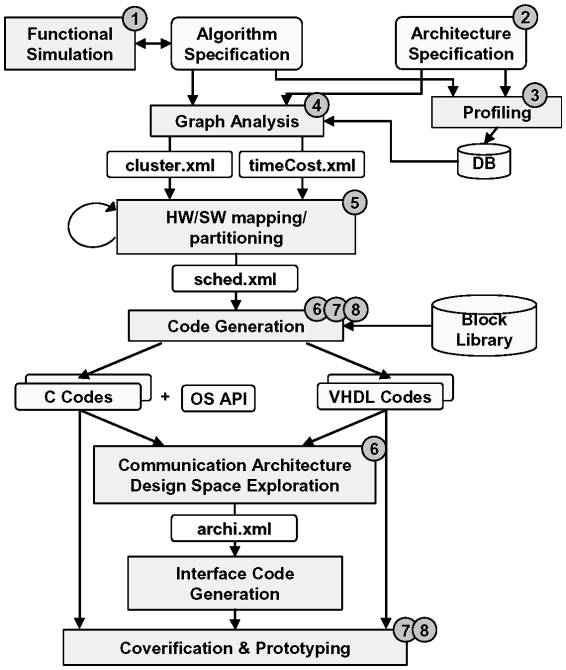

Figure 2.6: Design flow of the Peace framework. Figure coming from [16].

ming language or model but who are used to numerical computation tools, such as mathematicians. However the fact that is necessary to provide an XML architecture description in input as well, which requires a bit of design knowledge to fill, kind of neutralize this advantage. In addition hArtes targets a specific type of architecture: the MOLEN paradigm, which is thus limiting the design possibilities.

2.2.5 PeaCE

PeaCE [16] is a codesign tool for the design of systems for multimedia applications, developed by the CAP laboratory at Seoul National University. It is based on the Ptolemy framework [17], which is used for the design of real-time framework through modeling through extensive use of several models of computation and simulation. It starts from a model of an application using three different Model of Computation (MoC)s: one based on Synchronous DataFlow, called Synchronous Piggybacked Data Flow (SPDF) for the specification of computing tasks, another based on Finite State Machine (FSM), called flexible FSM (fFSM) for the specification of control task and another one, called task level-model that specifies high-level interactions between tasks. It also requires a set of the available architecture components (processors and IPs) to be used during the exploration step. The functional specification is used to generate a C code for functional simulation. Then for each task, performance measurements are made for each different processing unit with an ISS.

Then the first step of the DSE is launched: it explores the hardware solution, the partitioning and the mapping. Then code is generated for a hardware/software

cosimulation in order to get memory traces of the execution. These memory traces are then used during the second phase of the DSE where communication solutions are explored. In [16], only buses are considered as solution, so the DSE decides the number of buses, which bus is linked to which processing element and memory, bus frequency, etc. Finally, when a solution that satisfies the constraints has been found, implementation codes are generated for simulation tools and FPGA prototyping.

The advantages that PeaCE uses a model-driven approach to specify the input application by using and extending MoCs from the Ptolemy project, is also a limitation since designers have to learn how to model their applications with these three different MoCs instead of simply providing a C code, which is a widespread and well-known language among software designers. Moreover no exploration of the hardware acceleration is performed.

2.2.6 Xilinx XPS

Xilinx Platform Studio (XPS) [18] is a commercial design tool for systems targeting Xilinx FPGA devices. The design of the system can be made through the use of Graphical User Interface in order to add, remove and configure the elements of the hardware platform. It also provides several wizards to assist the designer in the creation of its system. It provides a library of proprietary IPs for the available bus and communication protocols, exploiting the board external peripherals. It also helps to integrate custom IP, through wizards and generation of interfaces in Hardware Description Language for integrating IP. The designed hardware can then be automatically transformed into a bitstream for the target FPGA through syntheses, mapping and placing and routing.

Although Xilinx recommends to use Xilinx *Software Development Kit* (SDK), a customization of the Eclipse framework, for managing the software aspect of the system, it is also possible to fully handle software project from XPS. This includes the compilation of the code as well as the specification of the drivers and the memory mapping.

While this tool, along with other Xilinx design tools, simplifies FPGA designs, it still remains a tool difficult to master and its numerous configuration options can often lead to bugs that can be hard to found/traced back at a later stage. The long synthesis times involved or the provided CABA simulation tool, *ModelSim*, do not allow to test quickly several designs and thus do not provide an adapted environment for design space exploration. So as such it should be used as a backend tool for implementation only.

2.2.7 Space CoDesign

Space Codesign [19] is a a commercial tool developed by SpaceStudio. Starting from an application specified in C code and split into tasks in order to express its parallelism and ease the mapping. Then the architecture must be described through a Graphical User Interface, this includes the number and types of processor, memories, buses along with the partitioning and mapping. Once the architecture specified, the architecture is estimated with the following criteria: performances, logic resources cost and power con-

sumption. This is done by generating SystemC TLM virtual platform of the hardware components and the C/C++ code of the software elements (tasks, RTOS). The generated virtual platforms includes non-intrusive profiling component for performances. The final design is selected based on the quality of results constraints provided by the designer and the corresponding hardware platform can be generated by another tool of SpaceStudio called GenX.

The limitation of Space Codesign is that, in spite of its GUI that allows to quickly modify the architecture and the partitioning, it does not perform any automated exploration.

2.2.8 Conclusion

Table 2.1 summarizes the presented ESL design frameworks. In the studied flow for H-MPSoC design, we can see that all of these tools failed to answer at least of the one of the problems stated in the introduction. For instance, Advanced Systembuilder, Space Codesign and Xilinx XPS do not perform any automated DSE, leaving to the designer the task to specify the evaluated architecture, which might be quite complex for a H-MPSoC. SystemCoDesigner and Daedalus are both too slow to be really efficient in an industrial context: the former because of its slow CABA-based performance estimation and the latter because of its exhaustive DSE. PeaCE has too complex input specifications and hArtes targets a specific kind of architecture, the MOLEN paradigm. Consequently there is a need for a new tool that will answers to these problems, by providing designers with a way to express their specifications with the level of details they wish, while automating the tedious aspects of design in order to provide quickly a satisfying implementation.

Table 2.1: Comparison of existing ESL frameworks.

	Inputs			**DSE**					**Outputs**		
	Soft	**Hard**	**Constraints**	**Automated**		**Scalable**	**Explored dimensions**	**Estimations**	**Code generation**	**Hard**	**Soft**
				SW & Arch	**HW IP**						
Ideal Framework	Widespread, well-known input languages / formalisms with no restriction	Architecture template specified with high-level formalisms	Provided by the designer through a GUI	Yes	Yes	Yes	Performance, area, power	Fast and accurate	Yes	Synthesizable Architecture	Adapted code
Us	C/C++ code in SANLP	Architecture template in AADL	Architecture template	Yes	Yes	Yes (pruning and constraints)	Architecture, Data & task mapping, Scheduling	Performance: Trace-based simulation / Cost: Model-based estimation	Yes	Synthesizable Arch.+ implementation files	Adapted code
Daedalus	C/C++ code in SANLP	Pearl Description language + XML file	XML files / GUI	Yes	No	Yes	Architecture, Mapping, Scheduling	Trace-based simulation	Yes	Synthesizable Arch.+ implementation files	Adapted C++
System-CoDesigner	Actor-oriented Model	N/A		Yes (MOEA)	No	Yes	Partitioning, mapping	Perf. with VPC	Yes	Bitstream	Adapted C++

Continued on next page

Table 2.1 – Continued from previous page

	Inputs			DSE					Outputs		
	Soft	**Hard**	**Constraints**	**Automated**		**Scalable**	**Explored dimensions**	**Estimations**	**Code generation**	**Hard**	**Soft**
				SW & Arch	**HW IP**						
Advanced-System-builder	Functional description (process + channel)	HW components + topology	Mapping constraints	No	No	N/A	N/A	FPGA prototyping: integration of HW profiler /Generation of CABA simulation files	Yes	HLS	C code + API for communication
hArtes	C code / Scilab language / Nu-Tech formalism	XML file	No	Yes	No	No	Application parallelism, mapping	SocLib performance estimation	Yes	HLS with Dwarv	Compiled binary
PeaCE	SPDF / fFSM / Task level models of computation	Set of available HW components		Yes	No	No	Two phases: first partitioning & mapping / second: communication	Cosimulation / FPGA prototyping	Yes, simulation + implementation	Synthesizable Architecture	C code + comm

Continued on next page

Table 2.1 – Continued from previous page

	Inputs			DSE					Outputs		
	Soft	**Hard**	**Constraints**	**Automated**		**Scalable**	**Explored dimensions**	**Estimations**	**Code generation**	**Hard**	**Soft**
				SW & Arch	**HW IP**						
Xilinx XPS	C/C++ code	VHDL component + library of IP		No	No	N/A	N/A	Implementation cost /Profiling solutions provided	No	Bitstream	Executable ELF
Space Codesign	C/C++ code split into tasks	XML file	Quality of Results	No	No	N/A	N/A	Cost, performance and power	Yes (with extra tools)	With extra tool (SpaceStudio GenX)	C/C++ adapted code

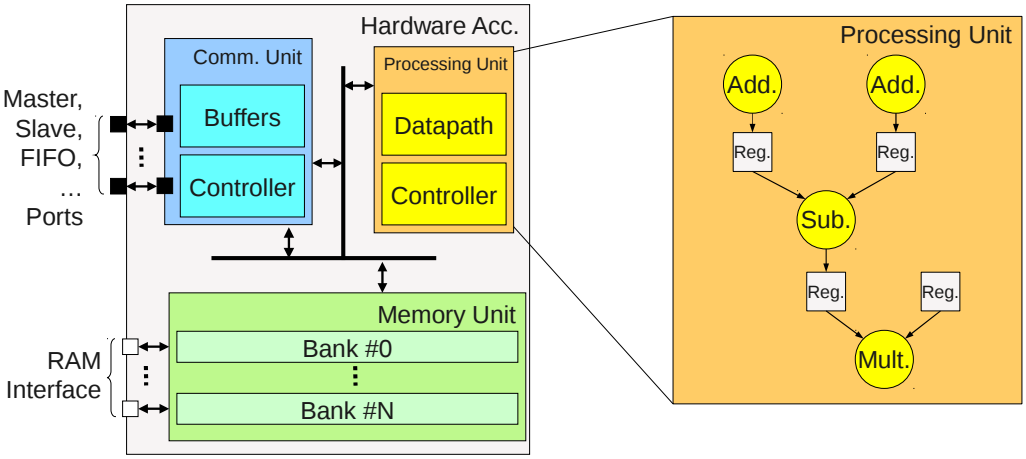

Figure 2.7: Architecture model of the hardware accelerator generated by GAUT.

2.3 High-Level Synthesis Tools

In H-MPSoC, hardware accelerators are used as a mean to speedup the system and it is part of the design space exploration to decide whether or not part of an application will be accelerated through hardware. If no off-the-shelf accelerator is available to the designer, it is possible to generate an IP through the use of HLS tool. It thus can be used as a tool in the ideal flow for H-MPSoC by providing synthesized on-the-fly hardware IP. In this section we present a few of the existing HLS tools.

2.3.1 Gaut

GAUT [20] is a high-level synthesis tool that can generate hardware accelerators from a behavioral C/C++ specification. It has been being developed for 15 years at ENSSAT Lannion and the Université de Bretagne-Sud Lab-STICC laboratory. The synthesis is performed under a time constraint according to:

- a generic architecture model as illustrated by Figure 2.7;
- a library of pre-characterized Functional Units (FU), i.e. basic operators such as adders, multipliers and hierarchically designed IPs;
- and a communication interface choice: FIFO or Ping-Pong memory.

The generated hardware block comes in different formalisms including VHDL and SystemC. Communication interfaces can be tuned to include specific features such as the Xilinx Fast Simplex Link (FSL) [21]. It is used in our framework to generate series of hardware IP during the hardware accelerators design space exploration which is detailed in Section 4.3. This tool is based on fast scheduling and allocation heuristics, it produces a RTL code compliant with a *register-to-register* architecture model. It is

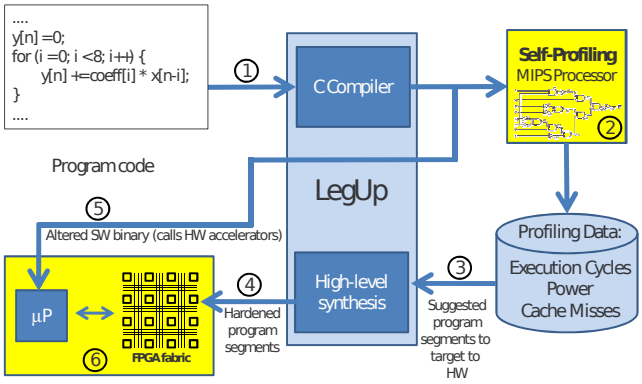

Figure 2.8: Design flow of the LegUp framework. Figure coming from [22].

decoupled from Logic Synthesis tools and has a linear computational complexity that we exploit to get fast accelerator evaluation within the whole H-MPSoC DSE loop.

2.3.2 LegUp

LegUp [22] is a research tool developed at the university of Toronto that semi automatically accelerates part of an application by generating and integrating a hardware accelerator into the design. It starts by performing an automated profiling of the application thanks to a modified MIPS processor-based FPGA implementation. The MIPS is modified in order to perform the profiling of its own execution. With this profiling, the parts of the applications to be accelerated through hardware are synthesized with LegUp behavioral synthesis tool. Then the software code of application is modified in order to call the generated hardware accelerators. Finally the software code is compiled and the hardware architecture is implemented on the FPGA, resulting in the implementation of a complete system.

LegUp is a quite complete tool for hardware acceleration of an application and is adapted to software designers that are not familiar with hardware design. However, LegUp does not actually performs any design space exploration: mapping and hardware decisions are taken manually by the designer, and the rest of the hardware architecture is determined prior to hardware synthesis.

2.3.3 C2H

The NIOS C2H compiler [23] is a tool that generates IP cores from a C specification. This tool provides an easy way to generate accelerators but does not perform any real high-level synthesis. Contrary to what is done by C2H, HLS means resources allocation and scheduling according to a predefined architecture model that also makes fast estimation possible. The synthesis from C2H or equivalent C-to-VHDL compilers could however be considered as additional solutions to feed the IP database used during DSE.

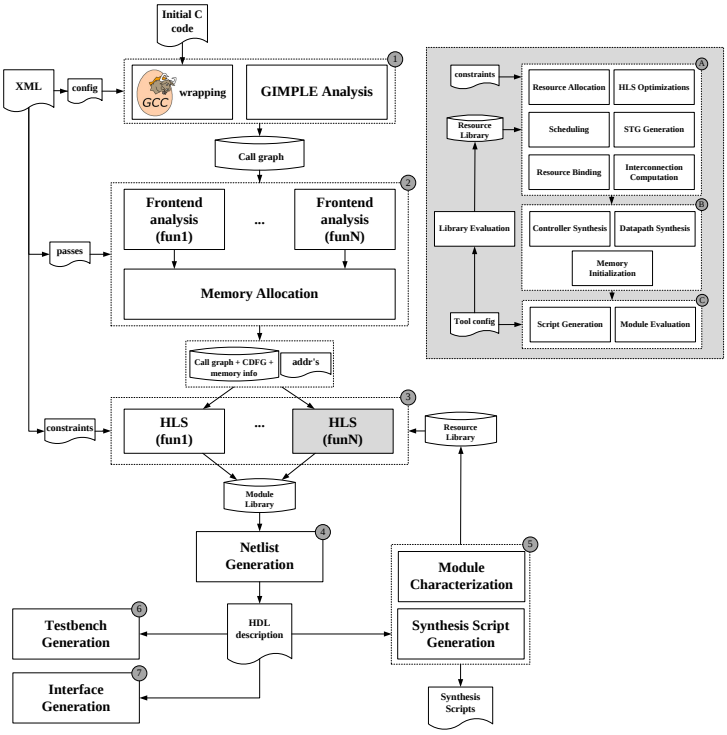

Figure 2.9: Design flow of Bambu. Figure coming from [25].

2.3.4 CyberWorkBench (CWB)

CyberWorkBench [24] is an impressive and complete but also commercial framework proposed by NEC, it relies on different HLS tools (data-dominated, control-dominated, control-flow intensive, ASIP design). This framework provides simulation and area/performance/power estimation tools for various FPGA technologies. The DSE tool is based on the modification of code (C/SystemC) annotation, such as for instance loop unrolling. However, we also observe that DSE includes low level logic synthesis tools that mean long synthesis time. Moreover the hardware/software mapping remains manual as well as the function/task/instruction parallelisms and the choice of I/O parallelism and protocols. This is a set of tools that do not exactly pursue the same objective as we aim to do, namely a fast exploration based at the task level on hardware/software partitioning and processor specialization and at the hardware accelerator level on the automatic exploration of instruction and I/O parallelism.

2.3.5 Bambu

Bambu [25] is a framework developed at Polimi which aims at the automated HLS of application. It takes as inputs the C code of an application and an XML file that specifies constraints and options of the flow. The input is transformed in a Static Single Assignment form through a modified version of the GCC front-end and undergoes a

series of simple optimizations, such as dead code removal, loop unrolling, constant propagation, etc. Then data are allocated on the available memories based on the application graph analysis and the constraints provided by the designer as inputs. Then, for each function of the C code, an HLS creates the functional hardware equivalent including datapath, controllers and memory interfaces. Several options are given to the designer for the HLS, offering tradeoff between area and performances. It supports generation of the implementation files of the final system, for several backend tools for FPGA (Xilinx, Altera...) as well as for ASIC. It can also generate testbenches for simulation tools and check that the synthesized system is functionally equivalent to the software application given as input.

Bambu is limited in the solutions it can provide as it can only generate full hardware system. This means that it does not deal with the software aspect of the system, thus leaving out of the exploration of data-parallelism and restricting the design possibilities offered to the designer and thus is not appropriate for the design of H-MPSoC.

2.3.6 AutoESL's AutoPilot

AutoPilot [26] is a high-level synthesis developed by AutoESL and now belonging to Xilinx[1]. Starting from high-level specifications it can provide the outputs RTL in VHDL, Verilog and SystemVerilog. It also provides a cost estimation of the FPGA logic resource as well performance — latency and throughput — at function and loop granularity which allow designers to optimize their code precisely.

One of the lack of AutoPilot is the impossibility to specify timing constraints on the input/output.

2.3.7 Conclusion

From this study of HLS tools, we can give a list that could be integrated into our framework to be used for hardware accelerators exploration (described in Section 4.3). The tools that can be used are: Legup, Bambu, AutoESL and Gaut. C2H cannot be used since strictly speaking, it does not perform HLS.

2.4 MDE-based Design of MPSoC

Several attempts have already been made to introduce Model-Driven Engineering into methodologies for MPSoC designs. We detail our usage of MDE in Chapter 5.

2.4.1 Model Driven Engineering for MPSoC DSE

In [27], a DSE methodology based on Model-Driven Engineering is proposed for MPSoC. Models are UML-based models and are implemented with the ModES framework which provides the necessary meta-models, as well as a transformation engine in order to

[1] which renamed it Vivado HLS.

translate model from one formalism to another. The models are divided into four domains:

- an application domain, which describes the tasks of the application
- a platform domain, which described the hardware components of the architecture
- a mapping domain, which describes the mapping between the task and the processors
- an implementation domain, which describes the info for synthesizing/implement the system

A library of components is provided to describe the hardware architecture. DSE is implemented as a heuristic based on simulated annealing which aims to minimize the following system characteristics: communication, memory, performance and power consumption. In order to perform evaluation of a design during DSE, it is necessary that the designer provide measurement of the performance, cost and power. For the software, this is done by compiling the code on the target architecture. DSE is performed by the H-Spex tools which explores the following characteristics for each processor: their numbers, the mapping of tasks onto them, the assignment of buses and its voltage. The components are stored in a repository with the characteristics necessary for evaluation of the system during DSE such as their size, their power consumption, their performance, etc. Once the DSE is over, a model of the final system is produced and through model transformation the final implementation files are generated. Their DSE is quite fast and accurate since in [27], it manages to evaluate a thousand designs in about one hour with very few error on the estimated values.

In this framework, MDE is used for model transformation, including the final implementation files, design-rules verification, and design space exploration. However they relies on the designer to provide all the necessary measurements values such as performance, cost, consumption for every hardware components and application task models. These values are quite important since the accuracy of the final implementation characteristics depends on the accuracy of the provided measurements. While they provide a technique for the software components, they are very evasive on how they get these measurements for hardware components, relying on the assumption that the component will be already present in the component repository.

2.4.2 Multilevel MPSoC Simulation using an MDE Approach

In [28] is presented an MDE-based DSE for MPSoC. The tool starts from description of a system, including thee application and the architecture, in the MARTE [29] formalism, which is an extension of UML for embedded and real-time systems. Then the DSE process is performed through successive simulations at different granularities. The simulation implementations are based on the SystemC language. At first a fast coarse granularity simulation (TLM PVT) is performed, checking for functional verification and contention detection, and allowing to quickly estimate a lot of solutions. Then,

from these estimations a selection is made which is then evaluated more accurately with a slower finer grained simulator (CABA). MDE is here used for automated model-to-model transformations, more specifically from the MARTE description to the different levels of simulation in SystemC.

While this presented methodology seems quite efficient for Design Space Exploration, it could be further enhanced by adding the possibility to generate the final implementation code of the system not just a simulation implementation. This would allow to have a more complete framework. Also, the authors provides no method to get the values (performance, cost, etc.) to characterize the models for evaluating a system during DSE, leaving this task to the designer. Furthermore, they consider only performance and power during DSE, and not logic resources cost, which can be problematic in an embedded system environment.

2.4.3 A Co-design Approach for Embedded System Modeling and Code Generation with UML and MARTE

In [30] and [31], a methodology is presented for the design of MPSoC. The models of both the application and the platform must be provided as MARTE and UML models. From this description, it possible to generate the synthesizable hardware components, whether through synthesis or through reuse of existing components. The project implementation files for Xilinx backend tools are also generated so that the project can directly be implemented on the FPGA target. Moreover it provides a way to model the dynamic comportment of a component thus allowing the take into account the dynamic reconfigurability of the FPGA. This allows the generation of system that uses dynamic reconfigurability and thus bring reconfiguration to novice designers.

The possibility to generate a complete from an abstract specification allows even designers unfamiliar with hardware design (typically, software engineers) to implement an hardware version of their systems. However the DSE is still a manual task that remains under the designer responsibility.

2.4.4 Conclusion

Among the studied MPSoC flows using MDE techniques for MPSoC design, one does not perform automated DSE, one does not generate the implementation files for the selected design and the last one does not provide any solution to get hardware characteristics values. So while there exists design tools that integrate MDE methods as part of their design flow, none of them are fully satisfying in regards to the problems stated in the introduction.

3
Flow of the Framework

In this chapter, the framework flow is introduced. First it presents the tool implementation and gives a rapid overview of the steps composing the design flow, from inputs to outputs through design space exploration and details some parts of the flow that do not represent major contributions of this thesis. The used formalism as well as the target architecture are explained. External tools that were used to perform some steps of the flow are introduced. Finally, we present how we used databases in order to favor reuse.

3.1 Flow Global Overview

This section presents the detailed implementation of our tool, the target architecture of the flow, its inputs and a general description of how it works.

3.1.1 Tool Implementation

Our framework is a combination of several tools and technologies. Its core has been written in Java language in order to benefit from its portability, its widespread use among software developers and its object-oriented paradigm. It was developed with the open-source IDE, Eclipse [32], which has a large and active community as well as numerous plugins to assist programmers in the development of applications. It also possesses several Java class libraries such as EMF which provides functionalities for the integration of MDE techniques.

Our framework also relies on external tools, some of which were modified in order to satisfy our needs. They are described in Section 3.2. In order to make the interfacing of these tools with our framework seamless, we use several techniques. For data exchange, the method used is through text files, whether formatted as plain text files, or in more formal specifications such as XML, depending on what formalism was accepted by the tools. Calls and control of the tools are performed through the use of Bash and C-Shell scripts.

3.1.2 Target Architecture

The output of the framework targets FPGA-based systems. FPGA are reconfigurable architectures, i.e. their computing functions can be modified after their conception and

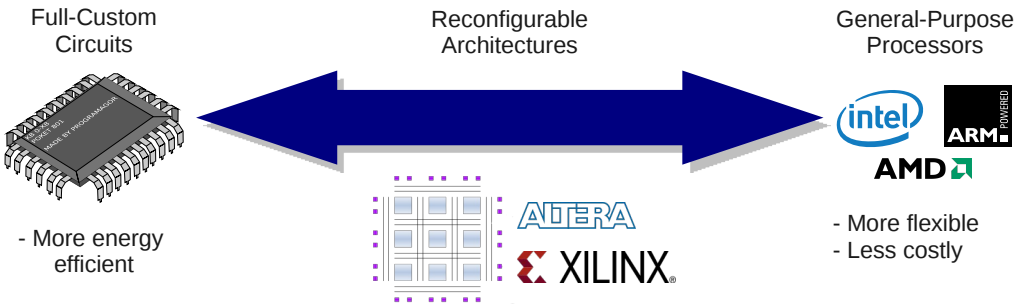

Figure 3.1: Spectrum of existing processing units types.

can even be dynamically reconfigured during execution. On the spectrum of processing units illustrated by Figure 3.1, FPGA is an intermediate solution between GPP, which can execute a wide variety of operations but with a low energy efficiency and Application Specific Integrated Circuit (ASIC) which can only perform the function they were built for, but with a very high efficiency and a very low energy consumption. The disadvantages of the ASIC are its high cost as well as its long design-time. So FPGA is an intermediate solution that provides the flexibility of the GPP while having a much greater energy efficiency. It is typically used for validating a design through prototyping, before it is put in production as an integrated circuit. It can also be used as an alternate cheaper solution to ASIC for circuits to be produced only in small volumes.

In spite of the advantages offered by FPGA, it is still not a widespread solution in the industry, for several reasons. First, the nonexistence of standards in FPGA design: there exists no common formalism that describes inputs and outputs with external peripherals and therefore no standard drivers, similar as what exists for instance for x86 architectures. This lack of standards makes difficult the reuse of design from one family of FPGA to another, increasing the design effort and the cost of porting a design. Second is the fact that FPGA design tools do not fully hide complexity to designers, making FPGA-target design a tedious, error-prone and thus costly process [3]. The presented framework aims at providing a solution to these problems, by rising the level of abstraction of the design and automating most of the design process thus providing a solution to the lack of standards.

In its current implementation, the framework produces project files for the Xilinx FPGA backend tools. However it is possible to modify the generation step to provide files for FPGA from other manufacturers. It is also possible to use the synthesizable code of the architecture to build an ASIC.

The architectures explored by our framework are Heterogeneous Multi-Processor Systems-on-Chip (H-MPSoC). Systems-on-Chip are systems where all the components (processing units, memories, buses) are gathered on a single chip. Multi-processor means that the system possesses several processing units, and thus can execute several

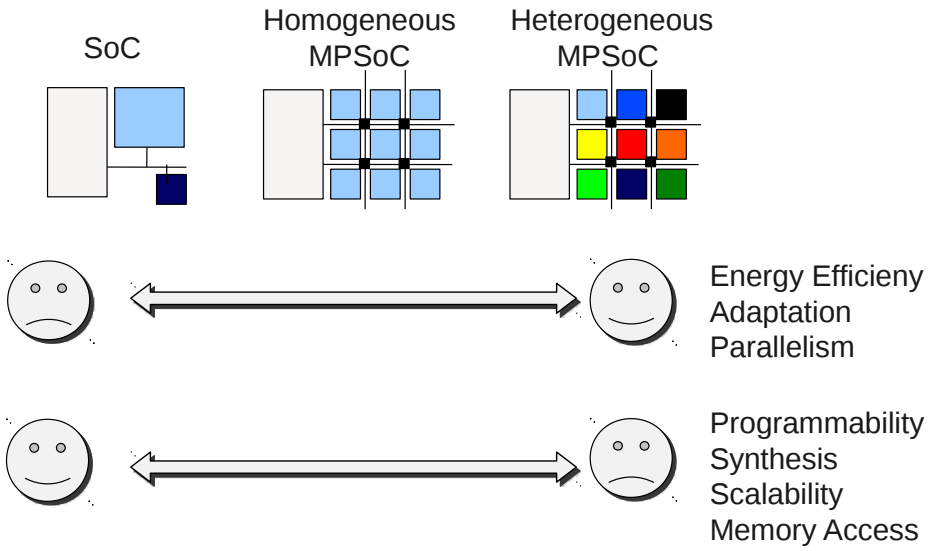

Figure 3.2: Pros and cons of the different types of System-on-Chip architectures.

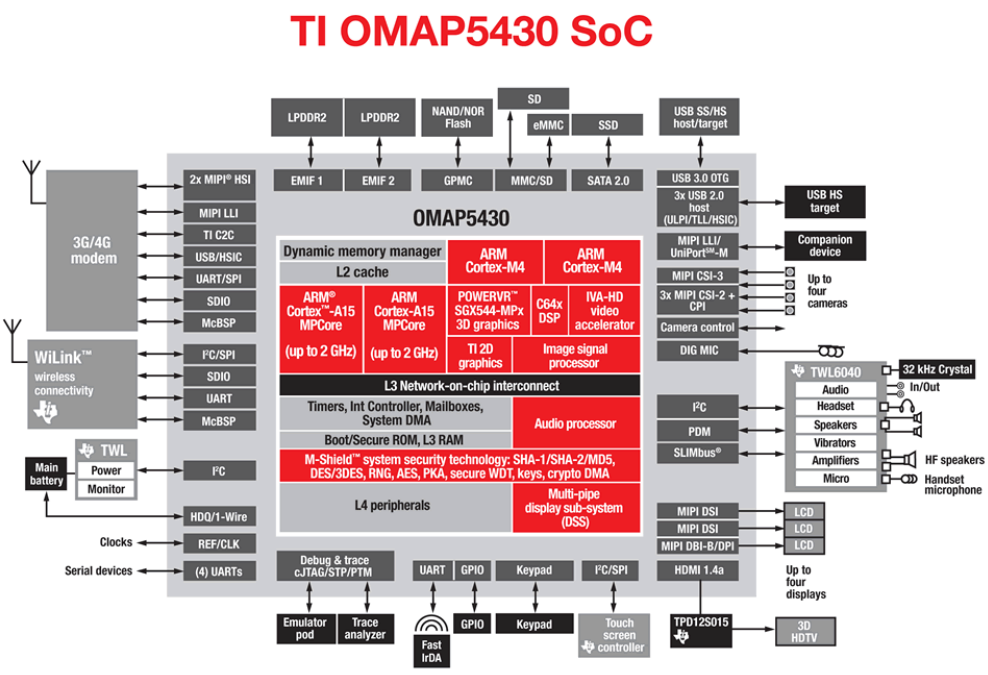

Figure 3.3: Example of a H-MPSoC: the Texas Instruments OMAP 5 architecture. Image: ©Texas Instruments.

operations in parallel. Heterogeneous means that these processing units do not all have the same nature: for instance, a system can have different models of GPPs coupled with dedicated hardware accelerators. This heterogeneity enhances the energy-efficiency of the system, since dedicated processing units allows for a faster execution of part of the application. A typical example of an H-MPSoC are found in smartphones which have on the same chip an ASIC for communication processing, an ASIC for video processing, and a GPP for other operations. Figure 3.3 shows the architecture of Texas Instruments OMAP 5 system-on-chip used in smartphones, which contains various processing units (two ARM Cortex A15 as GPP, a DSP, a GPU for 2D graphics and another for 3D, an ASIC for video encoding, etc.), memories and inputs/outputs modules that support several protocols and external peripherals.

The advantages and disadvantages of the different types of SoC are illustrated in Figure 3.2. For H-MPSoC, the advantages are:

- energy-efficiency thanks to the use of dedicated processing units which provide better performance and reduce energy consumption;
- multiprocessor parallelism allows to perform several operations simultaneously thus yielding better performance;
- adaptability since hardware specialization ensures the best performance possible for a specific task.

However H-MPSoC also have several disadvantages:

- They are difficult to program due to their heterogeneous nature which requires several skill sets to program. H-MPSoC has both hardware and software elements which belong to two different design domains that each requires their own tools and formalisms. Very few engineers actually have mastered both competences. This aspect makes it also difficult for designers from these two worlds to efficiently work together. Moreover their multiple processors make them also difficult to program since software programmers are taught to program in a sequential way and thus are not used to think in a parallel fashion.
- They are difficult to design due to the lack of metrics models. Their inherent complexity makes it hard to estimate the impact of a design decision on the performance, cost and power of the global system.
- They are difficult to optimize. In particular, memory accesses can be critical in multiprocessor systems since the number of processors increases the number of memory accesses and thus increasing the risk of congestion and contention both on memories and interconnect channels.
- They are not easily scalable since heterogeneity is an obstacle to task migration as a task implemented in hardware cannot easily be move back to GPP implementation.

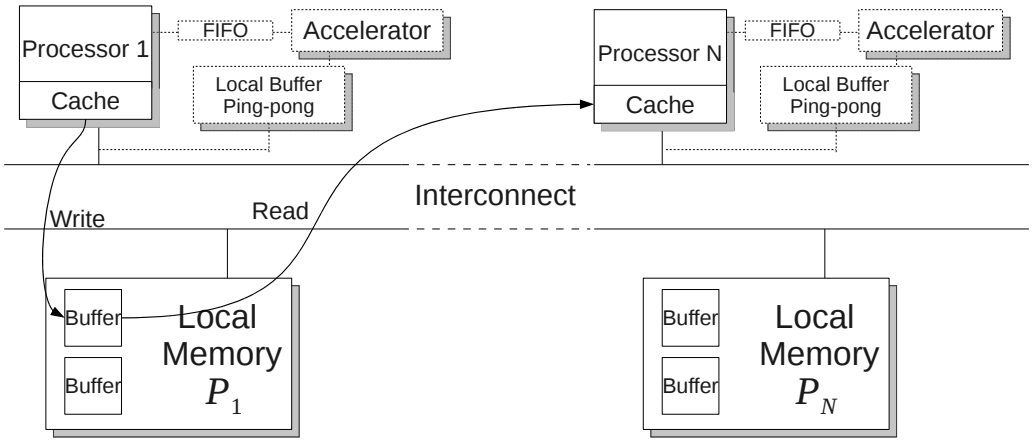

Figure 3.4: The model of architecture targeted by our framework.

All of the above reasons make absolutely necessary CAD tools to relieve designers from these difficulties. The availability of such tools is the key to usability of heterogeneous architectures.

Architecture Model

Our architecture model is basically a distributed heterogeneous multiprocessor architecture. It consists in a scalable H-MPSoC where each processing unit can potentially be associated with one or more of the following: a local memory, a shared memory or a coprocessor. The external peripherals and memories are reachable through an interconnect system that can be implemented as a multi-bus system or as a Network-on-Chip (NoC) [33]. The programming models are shared memory and message-passing. Inter-task communications are performed through a message-passing paradigm. Communications between processors and hardware accelerators can be implemented either in full FIFO, or with a mixed solution where control elements are send through FIFO and data are send with a Ping-Pong memory[1]. Communications between processors are implemented whether as software First-in, First-Out (FIFO) in a shared memory or as a Ping-Pong memory. Communications that go through data buffers are implemented as a zero-copy mechanism, i.e. processors are not processing the data to perform the copy from one buffer to another. An illustration of our architecture model is given in Figure 3.4.

[1]Ping-Pong memory is a technique where two memory buffers are alternatively read and written by two communicating tasks (e.g. a producer and a consumer): while the consumer reads data in one buffer, the producer writes its data in the other buffer and once both tasks have finished, buffers roles are reversed so that tasks do not have to wait for one another.

3.1.3 Inputs

As seen in Figure 3.6, the framework has two inputs:

1. A template of the architecture;
2. An application written in C language code.

Architecture Template

It is necessary to provide the design tool with a generic description of the architecture that can be used as a starting point for the exploration. It is also necessary to provide the components available for design exploration as well as some objectives and constraints to evaluate and bound the design space. Our architecture template provides these specifications for the DSE. The roles and structure of the template are detailed in Chapter 5. The template contains three levels of specifications. The first level specifies the domain-specific elements of the architecture. These specifications are static and thus represent parts of the architecture that remain constant throughout the DSE. The second level describes the constraints that define and bound the design space exploration, such as the minimum and maximum number of processors, the available types of processors, memories and buses, the cost constraints and performance objective, etc. The third level provides to the designer the possibility to specify a priori some of the parameters that would otherwise be decided by the framework during the DSE, such as enforcing a mapping decision. Such specifications allow for the designer to express his knowledge. In addition to guarantee good design decisions, the provided expertise also prunes the design space.

Application

In our framework, the application specification must be provided in C language code. Parallelism is considered at task level, instruction level parallelism is handled by processor/compiler in case of superscalar or Very Long Instruction Word (VLIW) processor or by HLS tools in case of hardware implementation. Designers have to choose the level of granularity for the task parallelism and accordingly manually split the application into tasks. Those are important decisions as they will impact the performance by providing more or less parallelism but they will also impact on the size and the frequency of the communications, which can represent a significant part of the execution time of the application. This action can be completed after the profiling step. Designers have to specify for each task if it is eligible to hardware acceleration or if it can be duplicated in order to exploit data-parallelism. These specification are given in the architecture template.

A MoC is the modeling of application following a specific formalism. Usually a MoC is adapted to model a specific domain of applications: real-time, DSP, multimedia, etc. Among the advantages of using a MoC is that it facilitates the automated processing of the application such as optimizations, simulation, synthesis, transformation, etc.

```
for (t = 0; t < 269; t++) {
    for (j = 0; j < 25; j++) {
            fetchprocess(&iqzz_d);
            iqzzprocess(&iqzz_d, &block_YCbCr);
            idctprocess(&block_YCbCr, &Idct_YCbC);
            yuvprocess(&Idct_YCbC, &pix);
            dispatchprocess(&pix);
    }
}
```

Figure 3.5: Example of a Static Affine Nested-Loop Program.

Since we target data stream applications, such as signal processing or audio/video processing, we have chosen Kahn Process Network (KPN) [34] as principal model of computation. In KPN, the application can be represented as a graph where processes are the nodes and communication channels are the edges. Communication are assumed to be performed through infinite FIFO channels. This means that writing on FIFO is a non-blocking operation while reading remains a blocking operation when no data is present in the FIFO. This property allows the execution of a KPN to be deterministic, i.e. the same inputs will always produce the same results. The transformation of the application into KPN is automatically performed by a tool developed at the University of Leiden/Amsterdam (see Section 3.2.2). This tool requires that the input application must be a Static Affine Nested-Loop Program (SANLP), meaning that the program must be in the form of one or more nested loops or conditional statements. The indices of the loops must evolve following an affine functions and the control of these indices must be static, i.e. they must be determined at design time and cannot evolve during execution. An example of a static affine nested-loop program is given in Figure 3.5.

Since it may not be always possible to be fully compliant to KPN formalism, it is possible to use a combination of other MoC for part of the program, as it is done in Ptolemy [17], but in this case the transformations must thus be made manually.

3.1.4 Flow Overview

The flow of the framework is illustrated by Figure 3.6. It starts with the inputs as described in the previous subsection, i.e. a C code as application specifications and an XML file for constraints and specifications (part A of the Figure 3.6).

Profilings & Parallelization

Then preliminary operations have to be performed on the application before starting the DSE (Part B of Figure 3.6). The application is first profiled with a profiler such as Gprof [35] in order to guide the designer into the task splitting of its application which is the next step of the flow as described in the above Subsection 3.1.3. Then

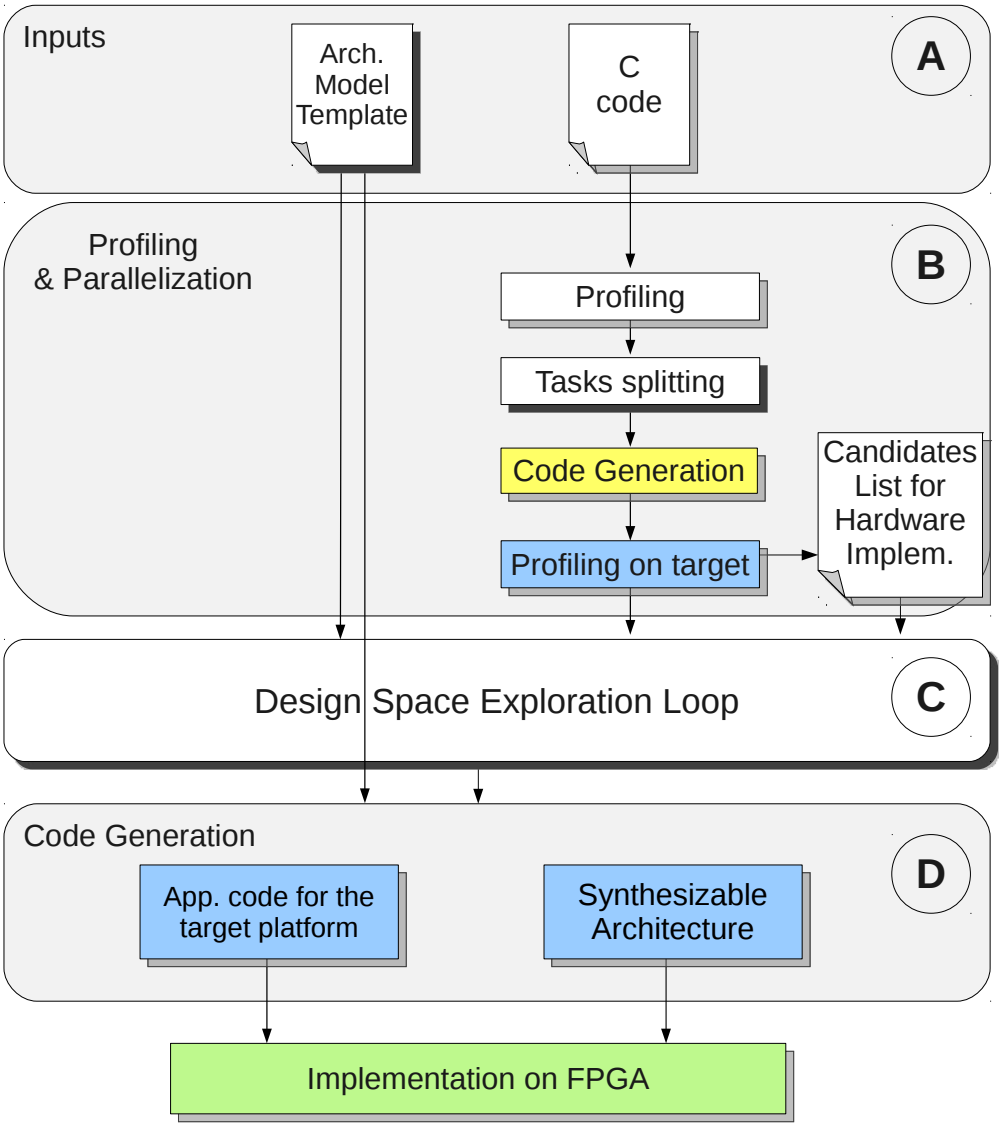

Figure 3.6: Overview of the framework flow. Our contributions are in blue, Daedalus is in yellow and external implementation tool in green.

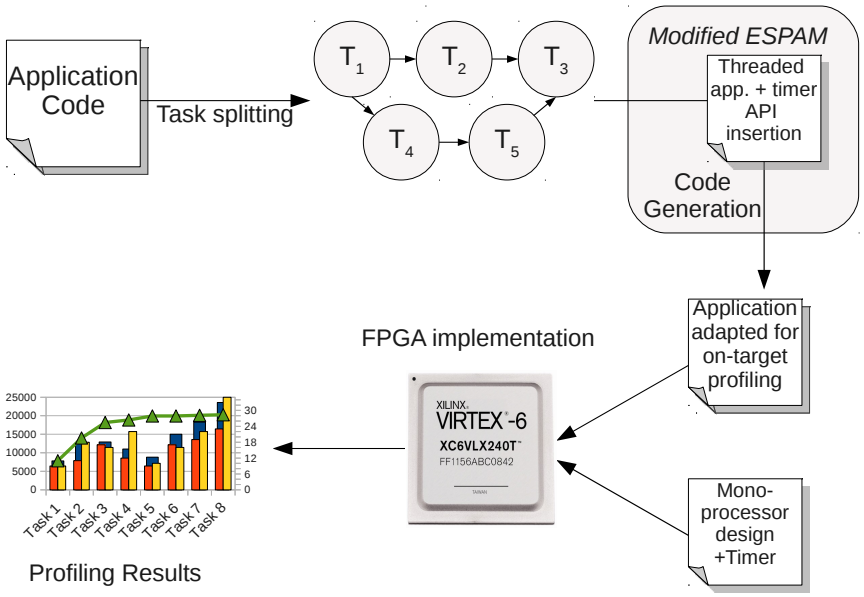

Figure 3.7: Flow of the profiling step.

the application is automatically transformed in order to perform an on-target profiling. Details of the on-target profiling are given in Section onTargetProfiling.

Design Space Exploration

Once this operation is over, the Design Space Exploration is launched (Part C of Figure 3.6). The algorithm of the DSE is explained in Chapter 4. During the DSE, decisions are taken about the hardware architecture of the system, the data-parallelism of the application, the task and data-mapping and scheduling. The output of the DSE is a selection of systems providing a tradeoff between area and performance to the designer.

Code Generation

Once the designer has chosen the final architecture, the code generation tool produces a synthesizable version of the selected hardware architecture, the adapted code of the application and the project files for the backend implementation tools (Part D of Figure 3.6). The generation implementation details are given in Section 5.6.

3.1.5 Automated Profiling

To be able to predict accurately performance during DSE, it is necessary to have accurate software execution times for each task of the application. In order to get the execution times, we perform an automated non-intrusive profiling on-target. This automated profiling step is illustrated by Figure 3.7. The application, after transformation into KPN, is automatically profiled through the use of hardware timers on a fully-soft

monoprocessor implementation. In order to perform this profiling a modified version of ESPAM generates an adapted code of the application. The generated implementation of the application is based on the Xilinx micro-kernel, Xilkernel [36]. Each task is transformed into a thread and calls to APIs that control the timer (*start*, *stop*, *reset*, etc.) are inserted in order to measure the execution time. A main file is also generated inside which the threads are initialized. The profiling results are then automatically collected through the standard output. They are written in a text file that will be parsed to be used later during DSE.

During the profiling, communication times are not measured since they are not relevant at this stage. Indeed communication performances depend on other factors such as data and task mappings, which are undecided at the time of the profiling. With this profiling we get the total number of cycles taken by one task to execute. This total is then divided by the number of times the task was executed in order to get the average number of cycles for one iteration of the task. During this phase, it is the responsibility of the designer to provide a set of data as input that is representative of real-life cases. Such set of data could represent a worst case in order to set the constraints in a way that will ensure that the output design will be able to handle all the encountered cases. For instance in video decoder design, there exists a standard set of videos acting as test cases to validate the decoder, which must be able to decode by respecting quality of service and performance (24 frames per seconds) constraints.

We also need to get the sizes of the data exchanges between the tasks in order to take into account communications in the performance estimations. This is done through the generation of a C code which prints the size of each data type used in communication through the use of the operator *sizeof*. These sizes are then written into a text file which is parsed by our tool.

3.2 External Tools

The design process of an ESL framework covers many research aspects: performance and cost optimization, communication, automated transformation, mapping, hardware and software modeling, power consumption, etc. For all of these research fields, lot of work has already been done and standards formalisms as well as tools have been developed to deal with those problems. Since it is useless to redeveloped already proven tools, the key issue was to select the tools that fitted best to our requirements and to set up an efficient tool flow that would be modular and that would seamlessly integrate the different tools. Among these tools, one was needed for the generation of hardware coprocessors, one for performance estimation and one for the final implementation. The use of these tools is made seamless to the user through the use of scripts for control and intermediate text files for data exchange.

Another important aspect is modularity, so we also try to use as much as possible standard formalisms, such as XML, UML, etc., for exchanging data between tools. The used formalisms for communication with external tools are illustrated in Figure 3.8. This is useful since designers might want to use other tools they are more familiar

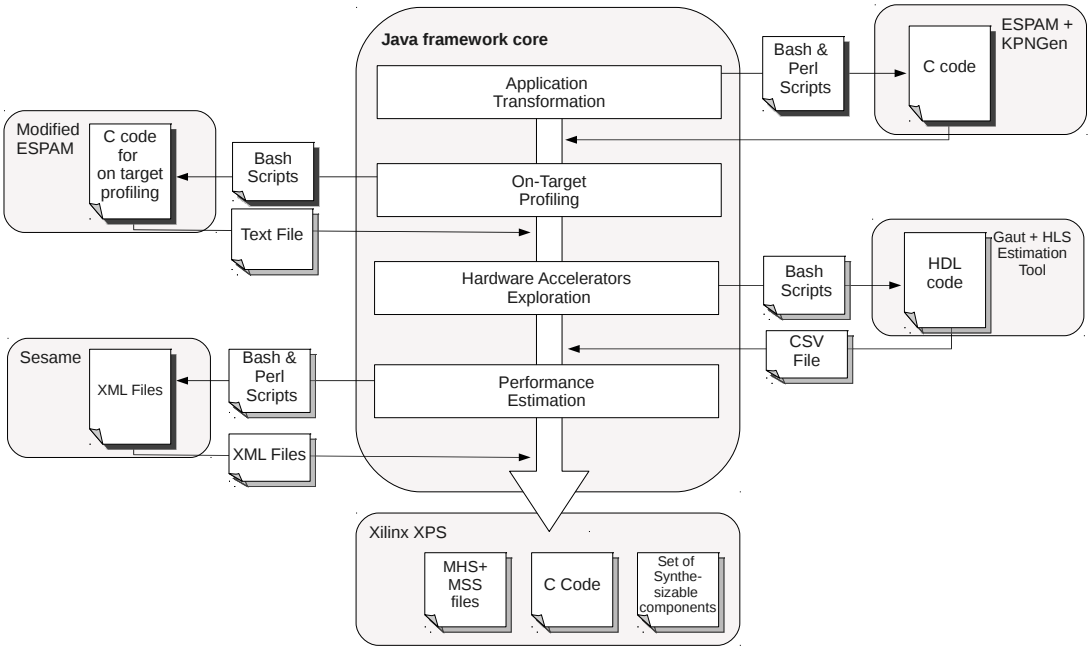

Figure 3.8: Technologies and formalisms used to interface with other tools.

with or that they judge more efficient, the use of standards insure that they are able to do so with minimal efforts. So in the rest of this section we present the tools we used and why we used them.

3.2.1 HLS Tool

We have chosen Gaut (see Section 2.3.1 for a detailed description) as HLS tool, since it fulfilled our requirements for exploring hardware accelerators. The interfacing is done through scripts that control the exploration and get the results. It is possible to use another HLS tool, provided that it can be controlled through command lines and that the model of architecture of the synthesized IPs is known in order to be able to perform cost estimations.

3.2.2 Daedalus

Since Daedalus [4] is a framework composed of several underlying tools, the following descriptions of these tools rely on how they work inside the Daedalus framework and explain how we adapted them to our framework. An overview of the design flow of Daedalus is given in Subsection 2.2.1. In Daedalus, those tools are linked together through the use of intermediate text files with different formalisms, mostly subsets of XML, and are called in a seamless flow made of a set of Bash and Perl scripts. We thus have modified some of the tools and scripts in order to insert our own tool and

contributions.

KPNGen

KPNGen [5] is a compiling tool developed at the university of Leiden that is able to parallelize a certain kind of application from its sequential description. Starting from a C code of an application, it automatically transforms the application into a Kahn Process Network. There are restrictions on the input application, which has to be a SANLP. As mentioned before, this restriction is similar to the one of our framework since we used KPNGen to transform the program into a Kahn Process Network.

We use KPNGen as is, with no modification, in order to transform the input application code into a KPN representation that can then be used by the ESPAM code generation tool that provides the corresponding C code.

ESPAM

ESPAM [6] is a co-design tool developed at the university of Leiden and is part of the Daedalus framework. It takes as input three descriptions: a platform description describing the hardware architecture of the system in XML, an application specification which is the application described in a Kahn process network and a mapping specification which describes the assignment of the channels and processes onto the hardware components. ESPAM produces as outputs: an RTL version of the platform description and of IP cores, the adapted application code.

In Daedalus, it is used to generate the KPN version of the code. It is also used to generate Xilinx backend tools project files, so that the design generated by Daedalus can be implemented onto FPGA. Since we reuse the same input formalism and the same simulation tool as the one used in the Daedalus framework, we reuse ESPAM for the transformation of the code into KPN formalism. However we do not reuse the generation of the implementation files for the Xilinx backend tools, although that is one of our objectives. This is due to the fact that for the generation of these files ESPAM uses the visitor pattern, in which the code to generate is hard-coded according to the visited element. This is not a very flexible approach and it does not provide designers with a convenient way to adapt a design to another target since it would require to modify the code and recompile the ESPAM tool. We believe that the MDE approach we have chosen is more appropriate to generate these final implementation files.

We also needed a way to automatically adapt the code for the on-target profiling, by inserting APIs to control the profiling timers (*start*, *stop* and *reset*). So we modified the visitor of the code generator in order to insert the APIs at the right places in the application code so that the profiling would accurately measure the execution times as describe in Section 3.1.5. It allows to fully automatize the on-target profiling and thus have precise measurements with little to no-effort from the designer.

Sesame

Sesame [7] is a modeling and simulation tool developed at the university of Amsterdam for exploration and estimation performance. The modeling of the underlying architecture is based on a library of components described in the Pearl description language [37]. It described several classes of component which implement methods, and communicate through blocking message-passing interfaces. The modeling of the input application is done in Y-Chart Modeling Language (YML) [38], based on XML. It is used to described the application model, the architecture model and the mapping between the former and the latter.

The simulation is based on trace of events. So a first execution of the task-split application is performed during which a trace of the occurring events are recorded. These events can be of three kinds:

- *read*, which corresponds to a task performing one data-reading on a communication channel;
- *write*, which corresponds to a task performing one data-writing on a communication channel;
- *execute*, which corresponds to one execution of a task.

For each task, the execution performance of the simulated processors must be provided by the designer in configuration files. Then using the trace of events, a model specifying the hardware architecture and the performance of the processing units, Sesame simulates the execution of the application on platform applying several variations according to the designers constrains:

- the number and type of processors
- the mapping
- the scheduling

The result is a set of statistics for each of these variations. This set includes performance estimation given as a number of cycles, the percentage of utilization of each component, the number of reads and writes for each channel, etc.

In our flow, Sesame is used to evaluate scheduling and the final performance of the system, after that the architecture exploration, and data and task mapping have been decided. In Daedalus, the exploration performed by Sesame is exhaustive within the constraints set by the designer. This means that the trace-based estimation performance is potentially very time-consuming since there might be a great number of solutions to test. We thus have modified the Sesame so that it can take as inputs only the set of architectures, mappings and schedulings that were the results of our DSE process.

3.2.3 Xilinx XPS

Our framework targets Xilinx FPGA and thus is adapted for Xilinx design tools. *Xilinx Platform Studio* (XPS) is the design tool for FPGA designs. It allows to describe through a graphical user interface the hardware platform to be synthesized on the FPGA as well as specifying the various software applications that will run on it. Our framework generates the corresponding text files used by XPS (cf. Section 5.6). Among those files, there are:

- The .mhs (*Microprocessor Hardware Specification*) which describes the hardware platform, with its components and its parameters.
- The .mss (*Microprocessor Software Specification*) which describes the drivers, along with the software parameters.

While this tool greatly simplifies FPGA designs, it still contains numerous parameters and options that remain pitfalls and make difficult the mastering of such tool by non-expert designers or by designers that do not wish to waste their time configuring the tools. That is why automatizing the creation of project and its parametrization through the generations of the above-mentioned files is a condition to increase efficiency of FPGA design and thus reducing its cost. Our MDE approach allows to adapt to possible evolutions of these files or to target other manufacturers tools, since it would only be necessary to develop a new model corresponding to the new specification formalism.

3.3 Database-based Strategy

In system design, applications belonging to the same domain often use the same functions: for instance a video encoder is likely to perform a Discrete Cosine Transform (DCT) since it is a common operation in image and video compression. Consequently, when implementing a design requiring one of those functions, they will be re-explored at each DSE. To avoid the waste of time induced by the re-exploration, reuse is an efficient solution that provides already tested and validated components.

For this reason, we used a strategy that favors reuse by implementing three databases in our framework: one for storing the architecture templates, one for the generated hardware accelerators and another to store models of FPGA architectures. This has two advantages: first it increases productivity by allowing the reuse of previous works. Second, it allows to shorten the DSE time by providing metrics from previous runs thus avoiding the need to perform an evaluation. However reuse does not always provide the perfect solution. For instance, the available elements in a database may not have the characteristics that would be optimal for a specific application and thus might require modifications by the designer.

3.3.1 Template Architecture Database

The architecture template database is used to store the domain-specific designs. This database provides designers with a choice of partially pre-designed architectures to use

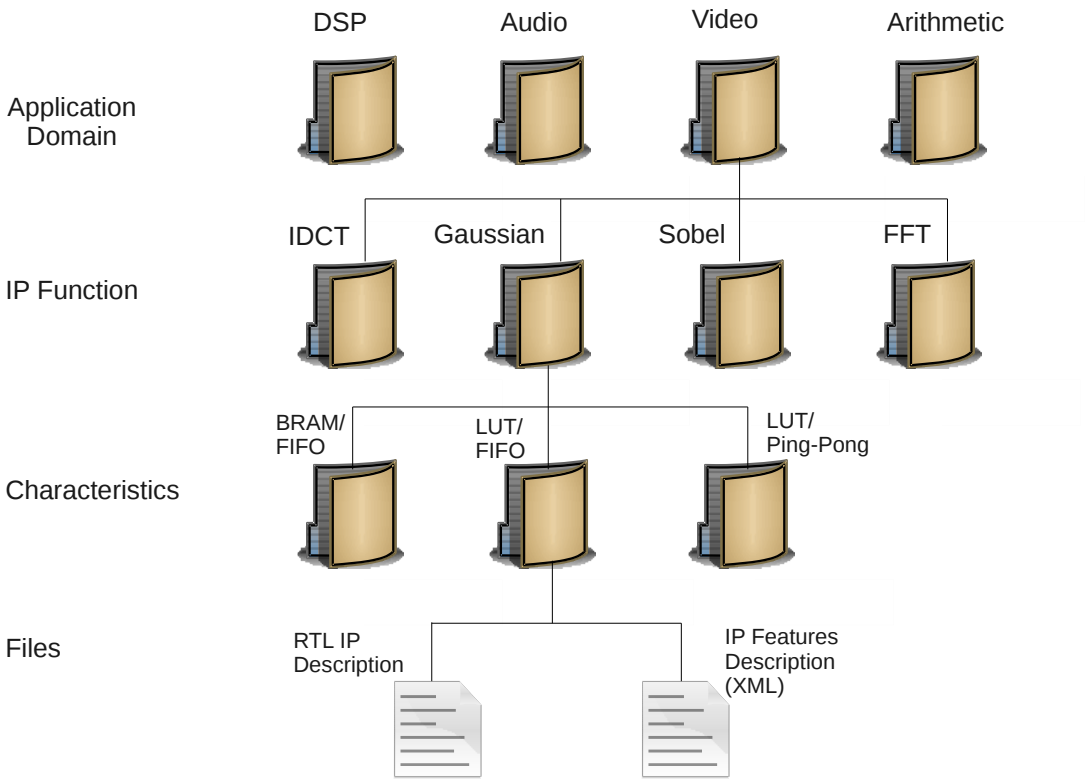

Figure 3.9: Organization of the hardware accelerators database.

as basis for their design. This template can then be customized by providing a set of specific constraints. Thus reuse is justified since for a system which would differ only slightly from an already-existing design or that would target another architecture, the template of the previously explored design can be taken and modified in order to fit the constraints of the new application. For instance, for a video decoder, an architecture for a new standard version (e.g. MJPEG AVC) can be efficiently designed as an evolution of a video decoder available in library.

3.3.2 Hardware Accelerators Database

The second database is the hardware accelerators database. It is used to store the result of the hardware accelerators exploration phase (see Section 4.3) and contains the RTL descriptions of various accelerators. For each accelerator, an XML file containing an estimation of its cost in logic resources is also stored so that it can be used by the DSE to estimate the area of a design solution. It is used during the hardware accelerator exploration step, where a check is performed in the database to see if a similar IP has already been synthesized, in order to avoid a costly exploration implying the HLS of a series of IPs.

The hardware accelerators database is implemented as a directory tree. There are

three levels of directory, illustrated by Figure 3.9 and organized as follow:

1. The first level is the general domain of application (Video, Audio, DSP, etc.).
2. The second level is the function of the IP (IDCT, Gaussian Filter, Sobel, etc.)
3. The characteristics of the IP (latency, bitwidth, etc.). These characteristics may not be all fully specified, depending on the operations that have been performed. For instance the cost estimations can differ if only the HLS estimation has been performed, if logic synthesis has been performed or if the implementation has been as far as place and route which would allow to specify the clock frequency.

Beside reuse of IP from previous runs of the tool, another solution to populate the database is to use IP libraries that provide free IP and under free [2] license, such as OpenCores [39]. This solution can be ideal for small companies that do not have the necessary fund to buy expensive off-the-shelf IP.

3.3.3 FPGA Model Database

For cost estimation of generated hardware accelerators (cf. Section 4.3), it is necessary to have models of the microarchitecture of the FPGA. The models are described in UML based upon FPGA technical specifications such as *Virtex-5 Family Overview* [40] and must be provided by the designer.

3.3.4 Reuse-based Strategy

These databases can be populated with elements from previous designs. In companies that means putting in common the design elements as the chance of reuse is likely to occur as a company is usually specialized in just a few design domains. It is also possible to imagine an open database hosted on a web-server that would be populated by designers willing to share freely their designs, in a similar fashion to free[3] software.

3.4 Conclusion

In this chapter, we have given a general presentation of our framework. We have specified the target architecture and its model, described the inputs formalisms, their restrictions and the transformations they undergo, and given a rapid description of our design flow steps relatively to the ideal flow presented in Chapter 2. We justified our reuse of tools, described how we seamlessly integrated them and potentially modified them to suit our needs. We also presented our strategy to favor reuse by relying on databases to store previous designs.

[2] As explained by Richard Stallman, here free is to be understood in the sense of "*free as in free speech, not as in free beer*". Although the former occurrence of free is to be taken in the latter sense.

[3] Again, as in free speech

4

Design Space Exploration Methodology

The Design Space Exploration (DSE) is the step where several design options are considered in order to find a suitable solution. During this phase, the evaluated design is modified within the boundaries specified by the designer's constraints. Depending on these constraints and on the DSE algorithm, several dimensions are explored. The dimensions are the characteristics of the system that are modified during exploration. Currently, the explored dimensions in our framework are:

- The number and type of processors.
- The number, type and size of memories.
- The use of coprocessors, as well as their cost and performance.
- The data and task mapping.

Modern systems-on-chip have grown so complex, that manual DSE is no longer possible and has become an automated process. Another consequence of the complexity is that exploring the design space in an exhaustive way is not possible for large boundaries. Scalability is thus an issue. Our goal is to offer to designers several options ranging from a greedy algorithm that will return the first solution that satisfies the constraints, an exhaustive exploration that will guarantee to get the optimal solution, or any intermediary solutions that will offer a tradeoff between the exploration time and the optimality of the resulting solution. Since designers are the most-knowledgeable persons about the designed system, it is necessary to take advantage of this knowledge. That is why our framework gives the possibility to designers to guide the DSE by specifying extra-constraints, and give them feedback about the evaluated design so that it can be used to refine the design.

In the next sections, we detail the implementation of the DSE steps in our framework.

4.1 DSE Algorithm

In order for the DSE to be able to operate efficiently and fast, several conditions have to be satisfied. We need:

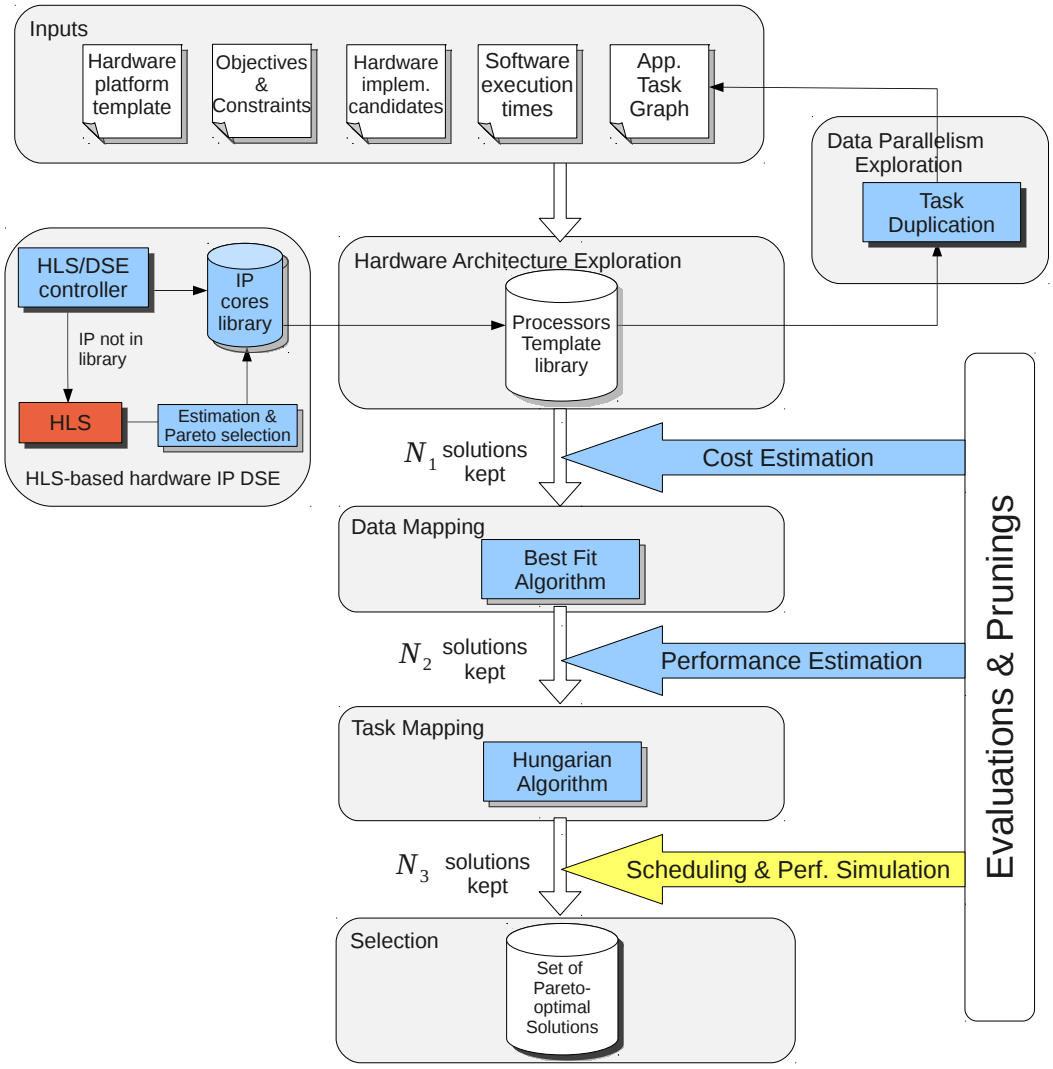

Figure 4.1: Flow of the Design Space Exploration.

- a well-defined model of architecture to provide a baseline for the exploration. This model is specified in the template given as input to our tool. Details of this template approach is given in Chapter 5;
- an accurate representation of the application specifications in a model of computation compliant with the application domain;
- the possibility to estimate the cost of an hardware acceleration: whether through pre-characterized IPs, or by being able to perform fast estimations through high-level synthesis;
- the ability to exploit the data parallelism of the application;
- efficient performance and cost estimator in order to be able to trim accordingly the design space even at early design stages;
- a variety of exploration options in order to cover a large area of the design space and minimize the risk of missing efficient solutions;
- the scalability of the DSE, by providing a possibility to prune the design space at strategic phase of the design;
- a way for the DSE to take advantage of the user knowledge of the designed system, by giving designers the possibility to express their expertise.

4.1.1 Algorithm

Figure 4.1 illustrates the flow of the Design Space Exploration strategy and a pseudo-code of the DSE algorithm is provided in Algorithm 1. It requires as inputs the application, the architecture template, the objectives and constraints set by the designer and the profiling data resulting of the previous step in the framework.

In order to be able to understand Algorithm 1, some definitions must be given beforehand:

- *AvailableProc* is the set of available types of processors (e.g. MicroBlaze, PowerPC, etc.).
- $PerfObjective$, $CostObjective$ are the performances and cost objectives respectively.
- $minProc$, $maxProc$ are the minimum and maximum numbers of possible processors in the design respectively.
- *SortedAccel* is a set of sets of hardware accelerators. Each subset contains accelerators for the same function with different latencies which are sorted by increasing area.
- *Solutions* is the set containing the currently selected architecture solutions during the exploration.

- *Architecture* is a representation of an architecture model with a set of hardware components, cost and performance evaluations.
- *NeedAccelerator* is the set of generated architectures, which do not meet the performance objective and consequently need hardware accelerations.
- $CandidateTasksToHW$ is the set of application tasks identified as good candidates to hardware implementation. In case of task duplication, a unique implementation is considered and all tasks instances are grouped ($GroupedTask$). Tasks are sorted by decreasing execution time which came from the earlier profiling step.
- N_1, N_2, N_3 and N_4 are values specifying the number of solutions that are kept between DSE stages. These parameters allow to balance the algorithm search-space/speed tradeoff and to favor some steps of the DSE over others. These variables provide scalability to the algorithm. For instance, if all $N_i = 1$, then it corresponds to a greedy algorithm while if all $N_i = \infty$ it means the exploration is exhaustive. N_1 is the number of selected architectures after the hardware exploration for a specific task, N_2 the number of selected architectures after the hardware exploration for all tasks, N_3 is the number of architectures selected during mapping exploration and N_4 is the final number of solutions that will be presented to the designer.
- T_{max} is an intermediate performance metrics for early evaluation of the architecture before the acceleration and mapping stages. It represents the worst computation time, where no cache is present in the architecture and consequently all data are read from the memory. This increases significantly the communication time and thus the total execution time:

$$T_{max} = \frac{ComputingTime}{NumberOfProc} + NWrite * WriteCost + NRead * CacheMissCost$$

  Where:

  - $NumberOfProc$ is the number of processors in the current architecture.
  - $ComputingTime$ is the total time given by the Sesame tool [7] for the mono-processor architecture. Basically, this is a software execution time but when a hardware accelerator is attached to the processor then the speedup is considered. This speedup estimation is first based on the profiling results that give the number of cycles used by each task. Then, this ratio is combined with the acceleration speedup provided by the hardware IP to compute the new computing time of task T_i. $ComputingTime$ is thus computed with the following formula:

$$ComputingTime = \sum_i ComputingTime(T_i)$$

where $ComputingTime(T_i)$ is the number of cycles for a software execution if T_i is not accelerated; otherwise:

$$ComputingTime(T_i) = \frac{\text{Number of cycles for a software execution}}{\text{Speedup obtained with the accelerator}}$$

- $NWrite$ is the total number of write accesses to the memory.
- $WriteCost$ is the cycle number taken for a write operation.
- $NRead$ is the total number of read accesses.
- $CacheMissCost$ is the penalty for cache miss as a number of cycles.

These values have been estimated once for the target architecture model with ad hoc profilings.

- Considering that T_{max} is a very pessimistic estimation, we set an α_0 parameter as a threshold set by the designer to bound the search space. So for each architecture, an α value is computed following the formula:

$$\alpha = \frac{T_{max}}{PerfObjective}$$

If α is inferior to the α_0 threshold, then the architecture is kept as a solution satisfying the performance objective, otherwise it is tagged as a candidate to hardware acceleration.

4.1.2 Explanations

The DSE algorithm, described in Algorithm 1 and illustrated in Figure 4.1, starts by exhaustively generating all the possible hardware architectures by taking only into account the available GPP and the minimum and maximum boundaries for the number of GPP. All these architectures are then evaluated in terms of cost and performances. The architectures which have a cost higher than the cost objective are discarded since the subsequent steps of the algorithm do not improve the cost but only the performance. Among the remaining architectures those which achieve the performance objective are kept as solutions, and those which do not achieve it are tagged as needing acceleration, whether through hardware acceleration or through the exploitation of data-parallelism.

Then for the most time consuming task that was tagged as accelerable, a series of hardware accelerators is generated, resulting in a selection of accelerators that each provides a tradeoff between cost and performance. These accelerators are sorted by increasing area, which correspond to decreasing latency since we only keep Pareto-optimal accelerators among the generated ones. The sorted accelerators are tested in that particular order until an accelerator that achieves the performance objective is found, which is thus the smallest accelerator satisfying the performance constraint. If no accelerator can provide sufficient acceleration then the second most time consuming task undergoes the hardware accelerator exploration and the process keeps going on until

Algorithm 1 Algorithm of the Architecture DSE.

```
1: S0 = generateAllSWArch(AvailableProc, minProc, maxProc)
2: for all Architecture A in S0 do
3:    evaluatePerformancesAndCost(A)
4:    if A.cost > costObjective then
5:       discard(A)
6:    else if A.α < α0 then
7:       Solutions.add(A)
8:    else
9:       needAccelerator.add(A)
10:   end if
11: end for
12: for all GroupedTasks T in CandidateTasksToHW do
13:   sortedAccel = launchHWExploration(T) //Call to HLS Estimator
14:   for all Architecture A in needAccelerator do
15:      for all Accelerator acc in sortedAccel do
16:         A2 = A.add(acc)
17:         // Pruning: test of matching between Task Instance / Accelerator Number
18:         evaluatePerformancesAndCost(A2)
19:         if A2.cost > costObjective then
20:            discard(A2)
21:         else if A2.α < α0 then
22:            Solutions.add(A2)
23:         else
24:            needAccelerator.add(A2)
25:         end if
26:      end for
27:   end for
28:   keepNBestSolutions(needAccelerator, N1)
29: end for
30: keepNBestSolutions(Solutions, N2)
31: for all Architecture A in Solutions do
32:   generatedArchitectures = ExploreMappingAndScheduling(A) //cf. Alg.2
33:   for all Architecture A2 in generatedArchitectures do
34:      evaluatePerformancesWithSesame(A2)
35:      evaluateCost(A2)
36:   end for
37:   Solutions.add(keepNBestSolutions(generatedArchitectures, N3))
38: end for
39: keepNBestSolutions(Solutions, N4)
40: return Solutions
```

satisfying solutions are found or that no accelerable task remains. Another solution explored to speed up the application is to exploit data parallelism. By duplicating some of the most time-consuming tasks and mapping them onto different processing units, it is possible to perform in parallel an operation on different sets of data.

Once the set of hardware components of the architecture is decided, the mapping and scheduling of the tasks and their communications are explored. For this we implemented the Hungarian method [41], a combinatorial optimization algorithm for which the function cost takes into account the performance, communications and precedence of the tasks (more details in Section 4.6.2).

Also, at several stages in the DSE, pruning are made to discard uninteresting solutions and to keep the size of the exploration space within reasonable bounds. A first pruning occurs during data-parallelism exploration, by not considering architectures that have fewer processing units than the number of time tasks are duplicated. A second pruning takes place after scheduling evaluation, where for each architecture only the mapping/scheduling combination giving the best performance is kept.

4.2 Performance & Cost Estimation

An efficient DSE requires accurate evaluations of performance of a given architecture, even and especially at early stages of the design. Some preliminary measurements are mandatory to compute these estimations:

- For each task, the number of cycles necessary for the execution of one iteration is obtained through a profiling on the target processor (MicroBlaze, PowerPC, etc.) as previously described in Section 3.1.5.

- Hardware accelerators, in our approach, are produced by HLS. The latency is obtained directly, since it is one of the HLS constraints.

- For buses and memories, their characteristics must be given by the designer as part of the inputs. These characteristics can generally be obtained through the manufacturer documentation and are specified in the input template.

With these values it is possible to have performance estimations that can be very accurate for a simple monoprocessor architecture that would execute the tasks sequentially. However in multiprocessor architectures, it is necessary to take into account additional factors such as the task and data parallelisms of the application, the potential resource contentions depending on communications, cache misses, mapping and scheduling.

In the next sections, we explore how the performance estimations evolve as the evaluated architectures become more refined as they go through the stages of the DSE algorithm.

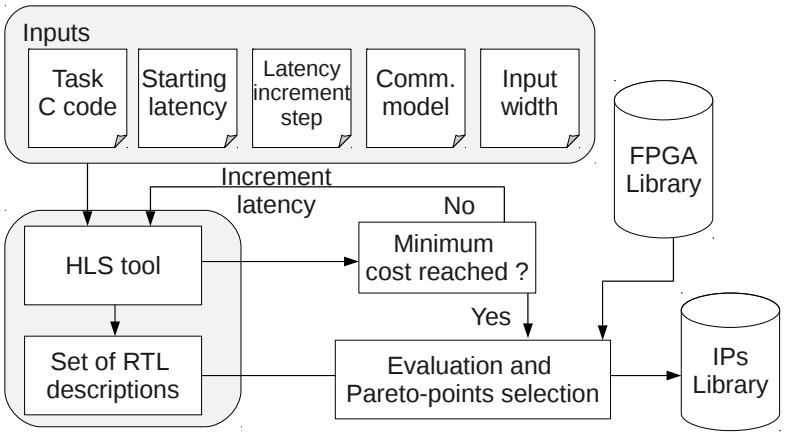

Figure 4.2: Flow of the Hardware IP exploration.

4.3 Hardware Accelerators Exploration

Parts of the work presented in this section were developed during Van-Trinh Hoang's Master thesis [42] that I co-supervised.

One of the contributions of the framework is the exploration of the hardware accelerators. In existing ESL design tools for H-MPSoC, there are usually two methods used to get hardware accelerators: IP libraries and on-the-fly high-level synthesis. However none of these methods seems to be the ideal solution.

IP libraries may not be able to provide a solution adapted to the specific needs of the designer, since the available IPs may have been designed for a different target and thus have been designed for a specific bus or may not have the correct bitwidth, etc. So it might be necessary to modify the IP in order to comply with the requirements of the system under design. Another problem is that commercial IP libraries might have a prohibitive cost in order to be used by small companies, even though there exist free IP cores libraries such as OpenCores [39].

On-the-fly synthesis is performed by providing a functional specification to an HLS tool that will generate a synthesizable version of the function. However when using on-the-fly synthesis, ESL design tools only perform one synthesis, thus making no exploration. A combination of both solutions is sometimes used: if the needed IP is not present in the library, then a synthesis is launched and the result is then added to library for future reuse.

In our framework, we perform a design space exploration of the hardware solutions. Consequently, we perform on-the-fly synthesis not of only one hardware accelerator but of a series of hardware accelerators, thus providing a tradeoff between performance and area.

Flow of the Exploration

The flow of our hardware IP exploration is described in Figure 4.2. During DSE, when a task is candidate to hardware acceleration, first the IP library is checked to see if an equivalent IP is already available. If not, a script is called that launches a set of behavioral syntheses. The script is called with the following parameters:

- the path to the C code to be accelerated;
- the minimal latency L_{min}, expressed in number of cycles, which served as the starting point of the exploration;
- an increasing step Δ_t for the latency;
- the model of communication. Currently only FIFO and Ping-Pong memory are available;
- the input/output width which is the bitwidth of the communication channels between the processor and the hardware accelerator.

During this generation phase, the script starts by calling the synthesis tool, GAUT [20]. The tool generates the IP corresponding to the minimal latency L_{min}. The tool then checks that the minimal cost has been reached, which means that no further gain can be made on the logic resource consumption by decreasing latency. This minimal cost corresponds to the state where the IP have exactly one of every necessary operators, and thus that no parallelism is possible. If the minimal cost has not been reach, then the latency is increased by the increasing step Δ_t and the next synthesis iteration is launched. The iterations keep going until the minimal cost is reached.

4.3.1 HLS-based Estimations

To evaluate efficiently the cost of an accelerator means to quickly estimate the logic resources it will consume. Our method used for resource estimations relies on high-level synthesis of the accelerator as opposed to logic synthesis which, while more accurate on the resource consumption estimation, is orders of magnitude slower than HLS to be performed. So the deal here is to trade little accuracy for huge computing time reduction. The speedup obtained with our method over logic synthesis is detailed in the results Section 6.2.

In order to provide estimations, it is necessary to build two models: one representing the target FPGA architecture (e.g. a Xilinx Virtex 5) and the other corresponding to the architecture of the generated accelerators. The microarchitecture model of the target is described in Figure 4.4. It contains the parameters necessary to perform an evaluation of the logic resource cost. For example, for an FPGA it provides the number of inputs of a LUT, the size of a DSP block, etc. The UML representation of the model for the RTL accelerators generated by GAUT is given in Figure 4.5. It provides the characteristics about the Finite State Machine (FSM), the number and size of registers,

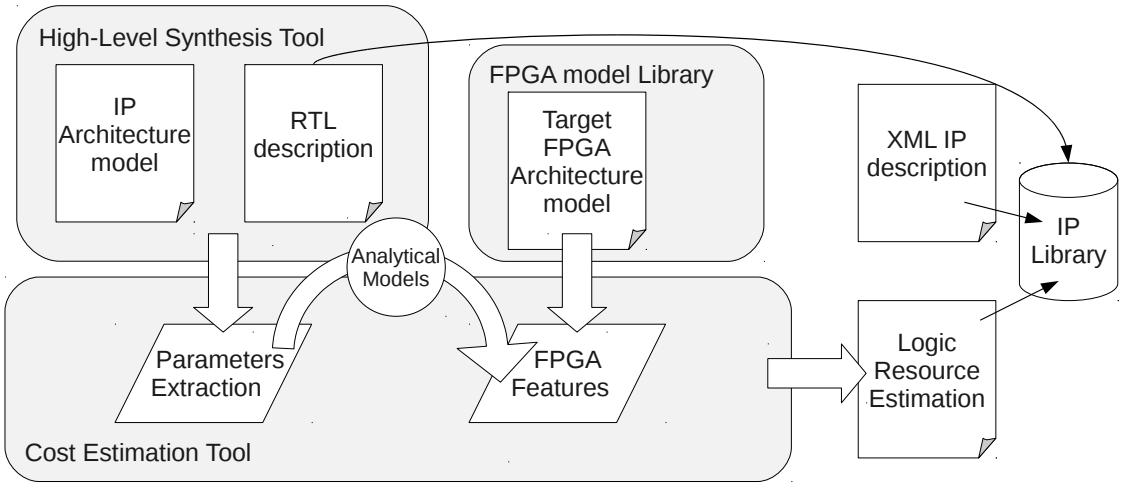

Figure 4.3: Logic resource estimation through HLS features projection.

the width and height of the FSM controller, the number and sizes of multiplexers, the width and size of memories, etc. Given these two models, the goal is to project the elements of the RTL IP description to the logic resources of the FPGA. For this, it is necessary to extract of set of analytical models that can predict accurately how the features of the IP will be transformed into logic resources on the FPGA target. The logic resources estimation process is illustrated by Figure 4.3. It begins by extracting the features from the IP, then using the analytical laws, it can compute an estimation of the consumed logic resources on the target.

For Gaut, the high-level synthesis tool we use, the architecture model of the hardware components generated is described in Section 2.3.1 and illustrated in Figure 2.7. It is made of operators coming from a library of pre-synthesized operators, characterized by a communication model (FIFO or ping-pong) and has a register to register architecture.

Since functional units are synthesized by GAUT using a library of operators generated through logic synthesis, the information on the consumed logic resources by functional units can be directly extracted from the numbers in this library. For Finite State Machines and communication interfaces, the logic resource cost depends on the implementation strategy (e.g. either in BRAM or in LUTs), and thus it is necessary to find the assignment methods used by the logic synthesizer in order to be able to estimate accurately the logic resources consumption. This method is thus flexible enough to work on several HLS tools and several targets, as long as the corresponding models can be provided.

To give an example of analytical model, we illustrate with the model of the cost of an FSM implemented in BRAM. The cost of an FSM implemented in BRAM is given

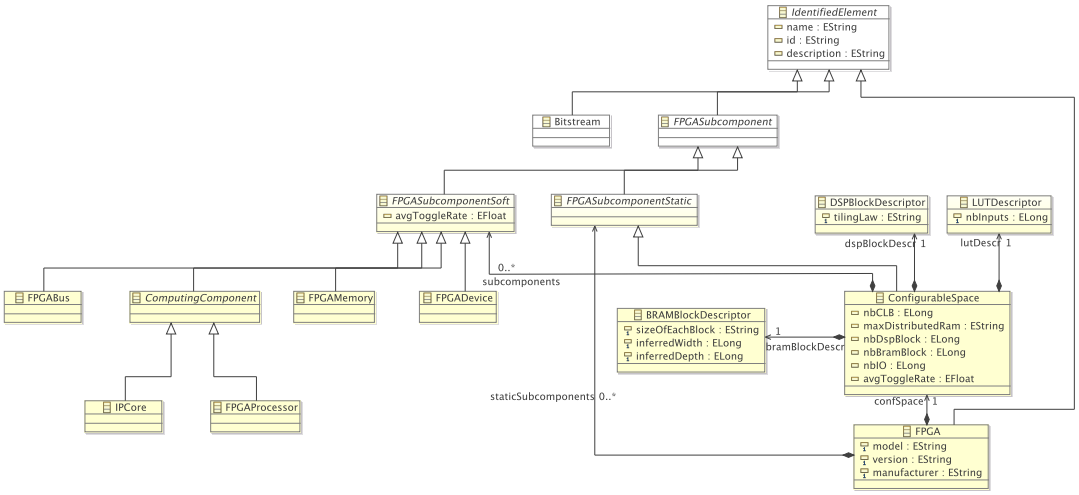

Figure 4.4: UML representation of the target FPGA metamodel.

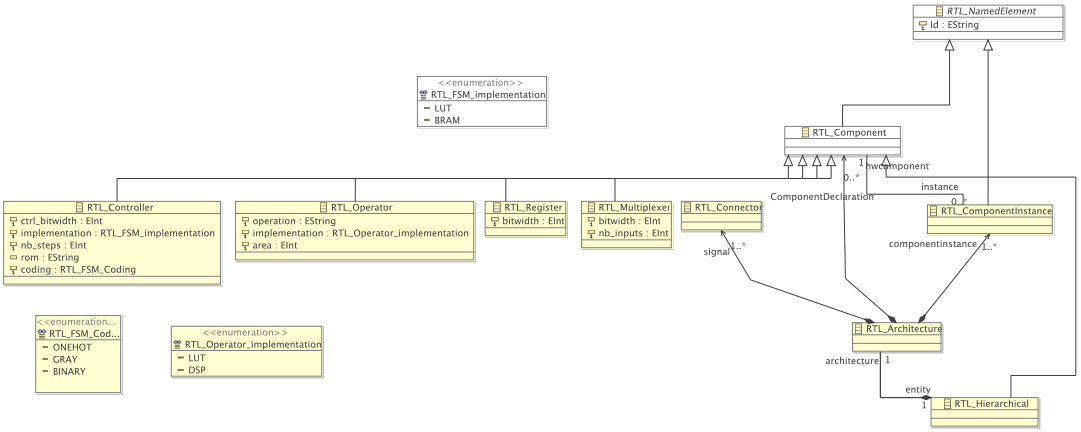

Figure 4.5: UML representation of the metamodel of an IP generated with GAUT.

by the following formula:

$$ConsumedBRAM = \left\lceil \frac{NbFSM_{states}}{BRAM_{depth}} \right\rceil * \left\lceil \frac{NbFSM_{command}}{BRAM_{width}} \right\rceil \quad (4.1)$$

The logic resource cost is then normalized in the form of slice estimation, a slice being the basic logic block in Xilinx Virtex FPGA family. Depending on the Virtex generation, a slice contains different elements. For example in a Virtex 5 [43], a slice contains four LUT, four one-bit registers, three multiplexers and one arithmetic logic. Thus we use slices as one of the units of measurement for estimating resource cost. However the final number of slices can be hard to predict since slices can either be used for their registers, for their LUTs or for both. It means that the use rate of slice resources is variable and depends on place and route decisions of the logic synthesis tool. So we compute an estimation interval with a lower bound S_L and an upper bound S_U. S_L corresponds to the ideal case where 100% of the slices are reused:

$$S_L = Max(Slices_{reg}, Slices_{LUT})$$

S_U corresponds to a theoretical ratio of reuse ($Ratio_{reused}$):

$$S_U = S_L + Min(Slices_{reg}, Slices_{LUT}) * (1 - Ratio_{reused})$$

The reused ratio can be tuned but, based on our experience, we found that the value giving the best results for our experiments was 0.8. With these limits, the frameworks provides a bounded interval of the consumed slices, which according to our experiments is quite accurate (see results in Section 6.2.1).

4.3.2 Pareto-optimal Selection

Once a series of accelerators has been synthesized and estimated, there are cases where some accelerators have a higher latency while being more costly than others with lower latency. This is due to the fact that when increasing latency, it is not always possible to remove an operator but it is then necessary to add some registers to store the stalled computed values in order to respect the latency constraint. Moreover a high latency also requires more FSM states, meaning more resources to store the FSM as given by the equation 4.1. Consequently the cost of the FSM ends up outgrowing the benefits of removing operators, as illustrated in Figure 4.6. This results in less-interesting IPs which allow us to make a selection, keeping only Pareto-optimal points. This selection provides an even shorter DSE, since less IPs have to be tested. An example for a small set of data is given in Figure 4.7. This Figure shows a series of IP generated for a Gaussian filter function. It compares our fast HLS-based estimation and the actual consumption obtained after logic synthesis. The vertical red lines indicate the Pareto optimal IP. More complete and detailed results for larger series of IP are given in Section 6.2.1.

The selected IPs are then added to the hardware accelerators library as described in Section 3.3.2, with the logic resource estimation cost. Later, it is possible to update

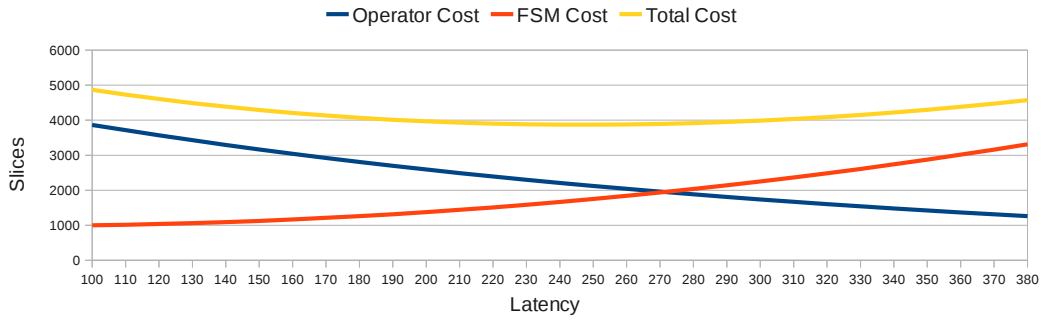

Figure 4.6: Illustration of the FSM cost/Operators cost tradeoff, showing that the FSM cost becomes more important than the decreasing of the operators cost.

the costs with the results of a logic synthesis, in order to have the real cost in logic resources if the IP is reused.

4.4 Data Parallelism Exploration through Task Duplication

Beside hardware accelerator, which is equivalent to instruction-level parallelism, another way to speedup an application execution is to exploit data-parallelism. Data-parallelism is the simultaneous processing of independent data as opposed to task parallelism which is performing different tasks simultaneously on different data. So when a task of the application is very time-consuming and cannot be accelerated through hardware, a solution is to duplicate the task, and map the duplicated task on a different processor in order to execute several instances of this task in parallel. Note that both acceleration solutions, hardware acceleration and data-parallelism, are not mutually exclusive as nothing prevents the duplication of hardware accelerators if the target logic resource capacities allow it.

Since task granularity is expressed manually by the designer during the task splitting phase, and is thus a result of the designer's expertise of the application, it is possible to automatize optimizations. The data-parallelism exploitation is made possible thanks to the KPN Model of Computation we use. And since task iterations in KPN are independent, data-parallelism can be exploited.

For a task that is duplicated, the potential acceleration gain depends on the number of times the task was duplicated. With the percentage of the total execution time taken by the duplicated task ($executionTimePercentage_{T_i}$), a rough estimation of the new execution time for the task T_i would be given by the formula:

$$NewExecutionTimePercentage_{T_i} = \frac{executionTimePercentage_{T_i}}{\text{Number of duplications}}$$

That formula corresponds to an ideal case, since it assumes that data can be provided fast enough to all the duplicated tasks so that there is never more than one of

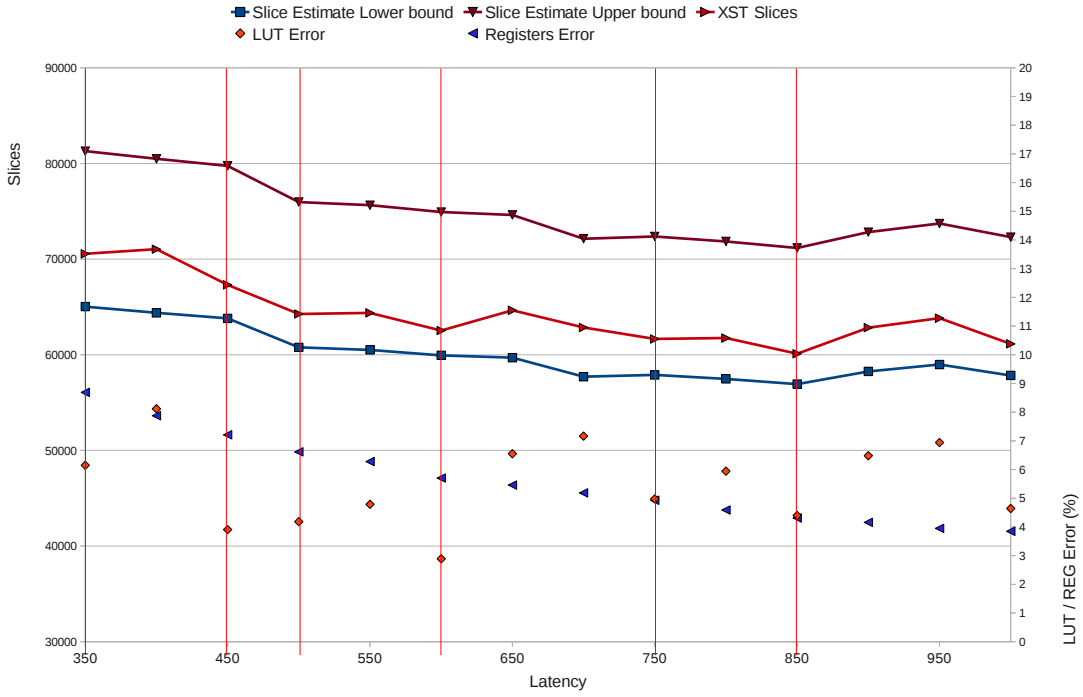

Figure 4.7: Results of a series of generated IPs for a Gaussian filter. The curves show the cost estimation and the actual cost, and the points are the percentage of errors in the estimates.

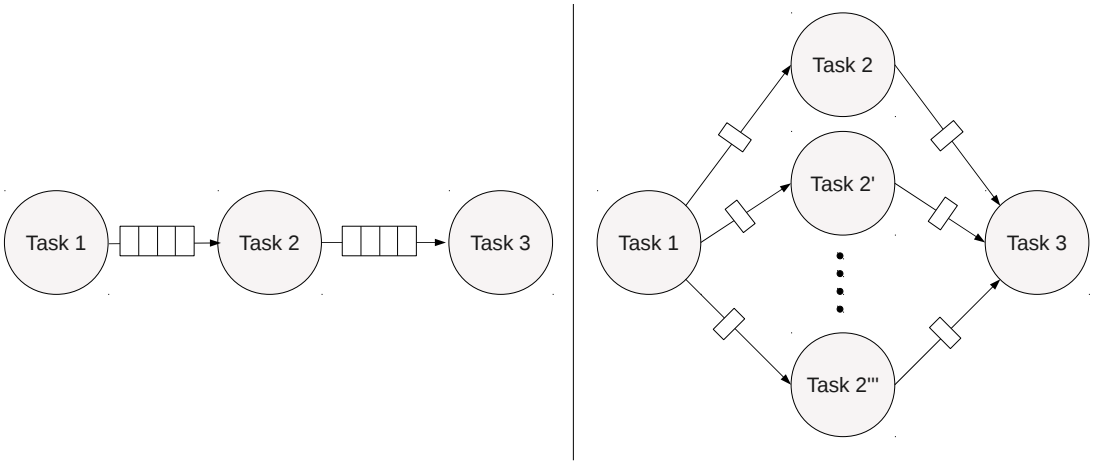

Figure 4.8: Task graph of an application before and after duplication. The transmitted data are also split equally between the different task, so that only the necessary are transfered

the duplicated tasks that is idle after the initialization phase. If we assume that the conditions for the previous formula to be true, then the global acceleration for the application provided by task duplication would follow Amdahl's law:

$$GlobalAcceleration = \frac{100}{(100 - executionTimePercentage_{T_i}) + NewExecutionTimePercentage_{T_i}}$$

The duplication of a task requires to modify the code of the application in order to dispatch the data to the multiple tasks and collect them back in the right order. This means that new read and write channels have to be instantiated and that the corresponding API calls must be added in the task code. In these newly instantiated channels only the data necessary for the task are sent, thus in best cases, there are no extra cost in communication due to data that would be send multiple times. Figure 4.8 shows the task graph of application before and after task duplication; Figure 4.9 shows the code differences in the task dispatching data before and after the duplication of a task four times. As can be seen, calls to other communication channels were added as well as conditions in order to equally dispatch data between the four duplicated tasks. Similar changes must also be made in the task collecting the data back.

Model versus Implementation

When duplicating a task, it is necessary to also duplicate the communication channels, and so make copies of data in the FIFOs, otherwise there would be multiple reads and writes on one channel from several tasks, which is forbidden by the KPN model of computation. This fact shows one of the limits in this model of computation: it does not allow shared memories which are convenient and efficient when communications

```
for(int c0 = 0; c0 <= 268; c0 += 1) {
   for(int c1 = 0; c1 <= 3; c1 += 1) {
      ports->IG_1.read(in_0ND_1);

      iqzzProcess(in_0ND_1, out_1ND_1);

      ports->OG_1.write(out_1ND_1);
   } // for c1
} // for c0
```

(a) Before duplication.

```
for(int c0 = 0; c0 <= 268; c0 += 1) {
   for(int c1 = 0; c1 <= 3; c1 += 1) {
      ports->IG_1.read(in_0ND_1);

      iqzzProcess(in_0ND_1, out_1ND_1);

      if(c1 == 0)
          {ports->OG_1.write(out_1ND_1);}
      if(c1-1 == 0)
          {ports->OG_2.write(out_1ND_1);}
      if(c1-2 == 0)
          {ports->OG_3.write(out_1ND_1);}
      if(c1-3 == 0)
          {ports->OG_4.write(out_1ND_1);}
   } // for c1
} // for c0
```

(b) After duplication.

Figure 4.9: Code of the loop of a task which sends data to a task duplicated four times. The code in red shows the differences between before and after adaptations were made for task duplication. Here each output channel receives data once every four iterations.

involve a large amount of data — e.g. a video frame. However it is important to make the distinction between model of computation and implementation. For instance, in order to avoid the extra cost in size due to the instantiation of duplicate communication channels, a designer might want to put only one copy of the data common to all the duplicate tasks in a shared memory. This is particularly true and possible with few modifications of the generated code when the size of the FIFO is equal to one — e.g. one frame. So to let designers express such an optimization decision, we have added the possibility to annotate the C code with pragmas specifying how the data should be mapped if a task was to be duplicated. This is part of the decisions that belong to the scope of designers.

Below is an example of a pragma annotation:

```
#pragma duplicateTask sharedMem #data1
```

The keyword *duplicateTask* means the pragma applies if the task is duplicated, the keyword *sharedMem* means the implementation of the data should be in shared memory and *#data1* is the parameter specifying the FIFO of the task that should be mapped in shared memory.

4.5 Communication & Memory Model

A major factor that has to be taken into account while exploring design are data exchanges. In some data-intensive applications, communication can be more time-consuming than the execution of the program itself. It is thus important to be able to accurately estimate the influence of task and data mappings on the overall performances. In order to estimate the time taken by the communications, models of

memories and buses are build. A memory is characterized by:

- its bandwidth, i.e. the maximum quantity of data that can get through in a given period of time. It is given in bits per second.
- its size which is the quantity of data it can store. It is given in Megabytes.
- the read and write burst latencies which are the numbers of cycles taken by the memory controller to prepare a burst of data for reading or writing.
- the number of read/written words per burst, which is the maximal quantity of data that can be transferred in a single burst.

With these values, it is possible to estimate the throughput capacity of both reading and writing operation of a memory for a given period of time. These values are used to check for potential congestion.

Similarly, to compute performance and check for congestion, buses are also described through the following characteristics:

- its bandwidth which is the maximal quantity of data that can be transferred in the bus in a given period of time. It is given in bits per second.
- the latency.

4.5.1 Congestion Detection

With the memory characteristics described in the previous section and the known quantities of data that go from one task to another, it is possible to compute if the buses and memories can provide enough resources so that no congestion occurs during the execution. For a memory and a given data mapping, it is possible to compute if memory can provide enough bandwidth. We first need to compute the number of necessary cycles to write the data of one iteration with this formula:

$$numberOfCyclesForWriting = writeLatency_{mem} * \frac{dataSize}{bytesPerBurst_{mem}} + dataSize \tag{4.2}$$

where:

- $writeLatency_{mem}$ is the write burst latency of the memory given in number of cycles;
- $bytesPerBurst_{mem}$ is the number of bytes that can be transferred through each burst;
- $dataSize$ is the quantity of data given in bytes that must be transferred at each iteration.

During a burst, a word[1] of data is written every cycle, however a burst requires a few cycles latency in order to prepare data for the coming burst. So the total number of cycles taken for one iteration is equal to *dataSize* cycles for writing data through bursts (one cycle per word), plus the latency penalty for each burst times the number of bursts necessary to transfer all the data. In a symmetrical way, the number of cycles for reading (*numberOfCyclesForReading*) is also done with an equivalent formula using the latency value for reading instead. So knowing these two values and the frequency at which the memory operates, the congestion of memory can be computed with this formula:

$$\text{congestionRatio}_{mem} = \frac{frequency_{mem}}{nbIterations * (\text{numberOfCyclesForWriting} + \text{numberOfCyclesForReading})} \tag{4.3}$$

where:

- *nbIterations* is the number of iterations that should be performed in one second to reach the performance objective;
- $frequency_{mem}$ is the number of clock cycles in a second, given in Hertz, which was provided by the designer as a constraint.

The equation 4.3 divides the available cycles in one second of execution by the total number of cycles necessary for reading and writing the mapped data in one second. This gives the occupation ratio of the resources. If it is over one then it means that the memory is overloaded and will not be able to provide data with a sufficient speed. If it is under one then in theory it should be able to provide data fast enough, however it is reasonable to have a security margin since communications on the bus may occurs simultaneously. During data mapping exploration, this information is used to know the degree of congestion of the memory and thus find a balanced mapping.

A similar formula can be used for communication channels:

$$congestionRatio_{comm} = \frac{frequency}{latency_{comm} * (dataSize / bandwidth_{comm})} \tag{4.4}$$

This gives the congestion ratio of the communication channels, thus providing an estimation on a particular data and task mapping, and provides an indication on the necessity to add extra communication channels to the design.

4.6 Data-Task Mapping & Scheduling Strategy

In this section we present the algorithm used to perform data and task mapping as well as the evaluation of the scheduling strategy. The whole decision process is described in Algorithm 2. Here is a few definitions necessary to understand the algorithm:

[1]The size of a word is dependent of the target architecture, thus this information must be provided with the specifications of the target architecture.

- $TaskClusters$ is the set of tasks to be mapped gathered as clusters of independent tasks.
- $ProcSet$ is the set of processors in the architecture.
- $DataSet$ is the set of data representing communication between two tasks sorted by decreasing size.
- $MemorySet$ is the set of memories sorted in decreasing order of speed.

4.6.1 Data Mapping

Data mapping is the assignment of the communications between tasks onto the memories of the architecture. Data mapping can have a high impact on the performance of the application: bad data mapping can lead to congestion or contention in buses and memories, thus leading to important performance degradation.

In our framework, the implemented data-mapping strategy follows a *best-fit* allocation which consists in assigning data to the fastest memory that has sufficient space to store the currently assigned data. The remaining available space on a memory is updated each time a data is assigned. Since we want to maximize the width of the explored design space, we generate several different data mappings in order to introduce diversity. To achieve this goal, we combine three strategies that insert interesting variations:

1. The read and write latency values of each type of memory are modified with a random coefficient. The interval taken by the coefficient is specified through two variables $coeff_{min}$ and $coeff_{max}$, which are respectively the minimum and the maximum values that can be taken by the coefficient (e.g. $coeff_{min} = 0.5$ and $coeff_{max} = 2$ will multiply the latency with a random coefficient that will give a result which lies between half the original value and its double).

2. Memories that are synthesized, such as BRAM, do not have a fixed size. So another dimension explored during data mapping is the size of such memories. The sizes that can be taken by a memory must be given as an input by the designer.

3. Last, the order in which data are mapped is considered in two ways: biggest data size mapped first and smallest data size mapped first. The intuitive strategy would be to map the biggest data on the fastest memory first, however the opposite strategy can provide better performance in some cases, similarly to As Soon As Possible (ASAP) and As Late As Possible (ALAP) schedulings.

Potential duplicate mappings are removed from the set of data mappings generated following these rules. The pseudo-code of the implementation of the algorithm can be seen in the data mapping part of Algorithm 2.

Algorithm 2 Data and Task Mapping Algorithm.

Initialization:
All hardware accelerated Tasks T are mapped on the corresponding $Accelerator_i$

for all N_2 architecture solutions **do**
  //**Data Mapping**
  //Consider several sizes for synthesized memories (e.g. BRAM)
  //Randomize memories latencies
  //Map biggest data on fastest memories first
  for all Data D in $DataSet$ **do**
    for all Memory M in $MemorySet$ **do**
      while M has enough space for D **do**
        mapDataOnMem(D, M)
      end while
    end for
  end for
  //Map smallest data on fastest memories first
  for all Data D in $ReverseDataSet$ **do**
    for all Memory M in $MemorySet$ **do**
      while M has enough space for D **do**
        mapDataOnMem(D, M)
      end while
    end for
  end for
  //Selection of the N_{2bis} best data-mappings

  //**Task Mapping**
  for all N_{2bis} selected mapping solutions **do**
    //Make first task mapping with less loaded processors
    for all Task T in TC_1, the first element of $TaskClusters$ **do**
      for all Processor P in ProcSet **do**
        if P is the less loaded proc **then**
          mapTaskOnProc(T, P)
        end if
      end for
    end for
    //Hungarian Algorithm
    for all Task cluster TC_i of $TaskClusters$ **do**
      for all Task T in TC_i **do**
        for all Processor P in $ProcSet$ **do**
          for all Memory M in $MemorySet$ **do**
            costMatrix = computeCostMatrix($\alpha \times execTime_{(T,P)}$, $\beta \times comm_{(T,M)}$, $\gamma \times procLoad_{(P)}$, $\delta \times memLoad_{(M)}$)
          end for
        end for
      end for
      applyHungarianAlgorithm(TC_i, costMatrix)
      checkForCongestion()
    end for
  end for
  //Selection of the N_{2ter} best task-mappings
end for

4.6.2 Task Mapping

The task mapping phase consists in determining the placement of each application task on the available processing units. The elements to be considered for an efficient mapping are:

- **Efficient computation**. Tasks should be assigned on processing units that are efficient to process them.
- **Minimal communication cost**. This a very important factor since communications can be the limiting factor over the process execution times. The rule followed to reduce communication cost is to mapped the task having the biggest data exchanges on the same processor so that they communicate only through the processor local memory, thus diminishing the global system communication load.
- **Load Balancing**. In data stream application tasks are dependent since a task relies on one or several previous tasks to get its data. So in order to exploit parallelism through pipelining, the load must be balanced between each processor. Indeed if there is one processor that has a more important load than the others then it becomes the limiting factor that drives the application global execution time.

The decision algorithm must thus takes into account all those factors in order to make efficient mapping decisions.

First Mapping Strategy

A first strategy that was experimented was to generate exhaustively every possible mappings for each different hardware architecture. This exhaustive set would then be evaluated through the Sesame trace-based simulation. As such solution could lead to an exponential number of mappings to be tested, we had a set of rules that trimmed the mapping set prior to evaluation by removing inefficient mappings and duplicates. The rules were the following:

- *R1:* If an architecture contains at least one accelerator, then the accelerated tasks must be mapped on the processor with the right accelerator.
- *R2:* If a task is duplicated then each duplicate task must be executed on a different processor in order to ensure that the parallelism is exploited.
- *R3:* Mappings must use all of the available processing units of the architecture, i.e. mappings where there is no task mapped on one or more processors are not considered.
- *R4:* Avoid equivalent mappings, i.e. for architectures that have several processors of the same kind (e.g. 2 MicroBlazes) then it is possible to have several mappings that are equivalent. For example, for an application with two tasks (T_1, T_2) and

an architecture with two similar MicroBlazes M_1, M_2, the mapping where T_1 is on M_1 and T_2 is on M_2 is equivalent to the one where T_1 is on M_2 and T_2 is on M_1.

These rules ensure that all the eliminated mappings are not optimal ones, with the exception of mappings that are optimal and equivalent. However while this method was providing good performance results, we have observed that even with the pruning rules, the mapping evaluations through the Sesame trace-based simulation was still consuming over 90% of the total DSE time. It was thus decided to use a method that would generate only a limited number of mappings for each hardware architecture, the challenge being that those mappings must be as efficient as possible.

Second Mapping Strategy

The method finally chosen for task mapping is a method based on the Hungarian algorithm [41], which can provide an optimal solution to the assignment problem in polynomial time. The pseudo-code of the task mapping is described in the task mapping part of Algorithm 2.

Hungarian Algorithm Example Here is a small example to illustrate how the Hungarian algorithm works. Let's consider the following cost matrix:

$$\begin{array}{c} \\ Task_1 \\ Task_2 \\ Task_3 \\ Task_4 \end{array} \begin{array}{c} \begin{array}{cccc} Proc_A & Proc_B & Proc_C & Proc_D \end{array} \\ \begin{pmatrix} 15 & 21 & 31 & 47 \\ 25 & 33 & 78 & 52 \\ 4 & 8 & 19 & 22 \\ 7 & 2 & 40 & 31 \end{pmatrix} \end{array}$$

First, it is necessary to find the minimum value of each row and subtract it to all elements in that row. It gives the following result:

$$\begin{array}{c} \\ Task_1 \\ Task_2 \\ Task_3 \\ Task_4 \end{array} \begin{array}{c} \begin{array}{cccc} Proc_A & Proc_B & Proc_C & Proc_D \end{array} \\ \begin{pmatrix} \mathbf{15} & 21 & 31 & 47 \\ \mathbf{25} & 33 & 78 & 52 \\ \mathbf{4} & 8 & 19 & 22 \\ 7 & \mathbf{2} & 40 & 31 \end{pmatrix} \end{array} \Rightarrow \begin{array}{c} \\ Task_1 \\ Task_2 \\ Task_3 \\ Task_4 \end{array} \begin{array}{c} \begin{array}{cccc} Proc_A & Proc_B & Proc_C & Proc_D \end{array} \\ \begin{pmatrix} 0 & 6 & 16 & 32 \\ 0 & 8 & 53 & 27 \\ 0 & 4 & 15 & 18 \\ 5 & 0 & 38 & 29 \end{pmatrix} \end{array}$$

Then the same operation is repeated for each column:

$$\begin{array}{c} \\ Task_1 \\ Task_2 \\ Task_3 \\ Task_4 \end{array} \begin{array}{c} \begin{array}{cccc} Proc_A & Proc_B & Proc_C & Proc_D \end{array} \\ \begin{pmatrix} \mathbf{0} & 6 & 16 & 32 \\ 0 & 8 & 53 & 27 \\ 0 & 4 & \mathbf{15} & \mathbf{18} \\ 5 & \mathbf{0} & 38 & 29 \end{pmatrix} \end{array} \Rightarrow \begin{array}{c} \\ Task_1 \\ Task_2 \\ Task_3 \\ Task_4 \end{array} \begin{array}{c} \begin{array}{cccc} Proc_A & Proc_B & Proc_C & Proc_D \end{array} \\ \begin{pmatrix} 0 & 6 & 1 & 14 \\ 0 & 8 & 38 & 9 \\ 0 & 4 & 0 & 0 \\ 5 & 0 & 23 & 11 \end{pmatrix} \end{array}$$

Then lines must be drawn over rows and column in order to cover all the zeros in a minimum number of lines. Here several solutions are possible but the minimum

number of lines is 3:

$$\begin{array}{c} \\ Task_1 \\ Task_2 \\ Task_3 \\ Task_4 \end{array} \begin{array}{c} \begin{array}{cccc} Proc_A & Proc_B & Proc_C & Proc_D \end{array} \\ \left(\begin{array}{cccc} 0 & 6 & 1 & 14 \\ 0 & 8 & 38 & 9 \\ 0 & 4 & 0 & 0 \\ 5 & 0 & 23 & 11 \end{array} \right) \end{array}$$

If the number of lines is smaller than the height of the matrix then the optimal solution has not been found yet. In such case, the minimal value of the elements that are not covered by a line must be found, and must then be subtracted to all elements not covered by a line and added to all values that are covered by both a vertical and horizontal lines:

$$\begin{array}{c} \\ Task_1 \\ Task_2 \\ Task_3 \\ Task_4 \end{array} \begin{array}{c} \begin{array}{cccc} Proc_A & Proc_B & Proc_C & Proc_D \end{array} \\ \left(\begin{array}{cccc} 0 & 6 & 0 & 13 \\ 0 & 8 & 37 & 8 \\ 1 & 5 & 0 & 0 \\ 5 & 0 & 22 & 10 \end{array} \right) \end{array}$$

This last step is repeated until the minimum number of lines is at least equal to the size of the matrix, which means that at least one optimal solution has been reached. The set of optimal solutions is thus made of all the combinations of tasks and processors that have a cost of 0, provided that each task is mapped on a different processor.

Implementation In our implementation, the cost matrix for the Hungarian algorithm represents the cost of the assignment of a task of the current task cluster on one of the available processors. Each cost is computed with the following formula:

$$(\alpha \times execTime_{(T,P)} + \beta \times comm_{(T,M)}) \times (\gamma \times procLoad_{(P)}) \times (\delta \times memLoad_{(M)}) \quad (4.5)$$

in which α, β, γ and δ are coefficients that can be set by designers to give more weight to the parameter(s) they wish to favor. The execution time and the communication time are used to determine the performance of a task mapped on that unit. $execTime_{(T,P)}$ is the number of cycles needed by the processor P to compute one iteration of the task T. $comm_{(T,M)}$ is the number of cycles necessary to transfer the data between the task T and the memory M. The other two parameters are used to prevent the apparition of behaviors that would result in bad performance: the processor load is used to keep the mapping balanced between the processing units; the memory load is used to avoid memory congestions and contentions. $procLoad_{(P)}$ is thus the load of the processor P, i.e. the percentage of the total computation time of the application taken by the processor P. $memLoad_{(M)}$ is the load of the memory based on its data transfer capacity, its frequency and the amount of data it should transfer in one second.

Once the task mapping is done, data that are mapped in "dynamic" memories, i.e. memories which are synthesized and which size is not static, such as BRAM, are

assigned to a processor. Depending on the task mapping, if data that corresponds to a communication between two tasks are mapped on the same processor, then the BRAM is implemented as a local memory of the processor (as seen as in Figure 5.2.b). Otherwise the BRAM is implemented as a shared memory associated with the processor that produces the data. Once all those steps are done, a check is performed that computes the load of the buses and other interconnects in order to detect possible congestions.

In order to maintain the algorithm scalable, the number of generated solutions at each step in the mapping is parametric. So, at the beginning of the data mapping there are N_2 architectures (cf. Algorithm 1, line 30), for each of which N_{2bis} data mapping will be generated and then for each of these N_{2bis} data mappings, N_{2ter} task mappings will be generated. At each of these mapping steps, it is possible to generate duplicate solutions which are removed at each selection step, so the total number of solutions at the end of the mapping algorithm is $\leq N_2 \times N_{2bis} \times N_{2ter}$.

4.6.3 Scheduling

Scheduling is the process that determines the order in which each task is executed. Several strategies exist, like *round-robin* which gives an equal share of execution time to each task, *first-come first-served* which gives access to the processing unit in the order the processes made their requests. In the current implementation of our framework, only the latter is available.

The performance estimation is then performed using the Sesame simulation tool [7] (cf. Section 3.2.2 for a description of Sesame and of the modifications we made). Our modified version of Sesame takes as inputs a list of architectures and their associated mappings, and, using an execution trace of the application, performs a performance estimation of the architectures. So the speed of this evaluation phase depends on two parameters. The first one is the number of iterations of the SANLP program that is used to generate the trace. The bigger the number of iterations, then the bigger the size of the trace and thus the bigger the time taken to simulate one architecture. It is thus necessary to choose wisely the size of application that will be used for the performance estimation. The second parameter is the number of mappings that needs to be evaluated.

The output is a performance evaluation of the candidate architectures which can be used by the designer to select the solution best-fitting his objective. For each evaluated architecture, only the mapping providing the best performance is selected. Alongside the performance, the cost in logic resources is provided as well so that the designer is given all the information needed to take the best decision.

4.7 Conclusion

In this chapter we have presented the decision algorithm for DSE. It provides an efficient and scalable way to explore several design dimensions and thus find a system

architecture satisfying the designers' constraints. This algorithm relies on fast and accurate estimation techniques for cost and performances at all the levels of the design process, from the early hardware architecture to the fully-mapped system in order to ensure that the best choices are made at each design step. We have also described how we widen the explored design space with the integration of a fast HLS-based exploration of hardware accelerators in the DSE loop. Another presented technique to increase the performances of the designed system is the exploration of data-parallelism through task duplication.

5 Template-based Approach

In this chapter, we introduce a template-based approach that simplifies design specifications and provides designer with a way to express design constraints that are the results of their skills and knowledge of the system. We also explain why and how we introduced MDE methods into our framework flow. These methods, coupled with the template approach, provides us with a way to enable design portability, to favor reuse, and to generate code.

5.1 Introduction to MDE

Model Driven Engineering comes from the software industry where it is used to provide specifications of an application system. It typically uses an abstract description of the system using a standard formalism, the most common being the Unified Modeling Language (UML). This abstract description can also be provided with a Domain Specific Language (DSL), which is a language specialized to describe a specific type of application.

The usual coupling in software industry is an association of UML and an object-oriented language such as Java since the object-oriented paradigm through its class and subclass representation fits well with modeling techniques. The UML representation can then be used to express properties and constraints in the program, as well as describing relations between the different entities of the program. From this UML representation it is possible to check that a design instantiating this model does not violate any of the specified constraints, thus ensuring the validity of a design.

Another possibility provided through modeling approach is the ability to generate an equivalent of the model in another formalism through code generation or model transformation. This transformation however requires that transformation rules must be defined. Once these transformation rules are defined, any correct instantiation of the model can have its equivalent generated in the targeted formalism. The advantage is that, if the transformation rules are correct, then the generated code is *correct-by-design*, and thus bug-free. This provides a huge gain in productivity, since it allows to skip costly tests and tedious debug steps. That is why we believe application of MDE methods should also be used in global system design including both software and hardware.

5.2 Application to FPGA-based Design

While FPGA is now a technology that has been around for over two decades, it is still not much used for embedded systems. This is in part due to the absence of standards in FPGA [3]. The lack of standards for the input/output of external peripherals of FPGA boards limits the design portability. So, to adapt a design from one FPGA board to another, it might be necessary to modify the communications with the external peripherals, implying the possibility of introducing new bugs in the system. The other barrier is the lack of standard processors and IP architectures including API that would facilitate the programming and the debugging of FPGA designs. Consequently, products based on FPGA require designers with high-level of expertise, which can be problematic for small companies which may not be willing to invest in such a risky and costly solution.

Thus using modeling techniques for FPGA designs is a solution to the lack of standards. By elevating the level of abstraction for specifications, it provides a representation that is independent from hardware specific details of the FPGA board and thus requires less specific knowledge from the designer. This independent model can then be used to generate the implementation files specific to the target FPGA through model transformation. This allows to greatly simplify the porting of a design from one FPGA to another, since the only requirement is to create just once the model for the new target in order to be able to generate the corresponding code, as opposed to the necessity to modify each design that needs to be ported.

Moreover it is also possible to exploit MDE verification and validation capacities, using model checking techniques to verify the correctness of a design. This is done by specifying a set of constraints in the model, that will be check by the tool to see if the modeled design meets the constraints. For instance it can be used to detect design error such as two communicating peripherals that do not have the same bitwidth. This can avoid errors that might otherwise have been difficult to find as design tools usually provides numerous options that can incompatible with each other [44].

5.3 AADL

Architecture Analysis & Design Language (AADL) [45] is a domain model language originally used in the avionics industry[1]. It has since evolved to a more global domain, to describe embedded systems, including the representation of both hardware (processors, memories, buses, devices) and software components (threads, processes, thread groups, data, subprograms). It has been used as the pivot language in the European project SPICES, which aimed at the development of methods and tools for the design of mission-critical embedded systems. It is also used in the Open-PEOPLE project [46] for the development of models for power consumption analysis of embedded system designs [47].

[1] AADL used to stand for *Avionics Architecture Description Language*.

We use AADL to model only the hardware parts of the system architecture, and thus only hardware components are considered. We do not need to model software, since its representation is given as a C code from which we perform all analyses and transformations. Each hardware component is described with two distinct types of declaration:

- a *type* that declares the external interfaces of the component through which can be connected to other components using input/output event and data ports, data and bus access, etc.;
- an *implementation* that represents the internal composition of the component. It can include one or several subcomponents.

A set of properties are predefined in the AADL standard to specify various component parameters. For example, for a generic memory component, the predefined properties are the size, the access rights, the word size, read and write times, etc. These predefined property sets can be extended by the addition of other property sets to model properties of specific components such as MicroBlaze processor [48]. An example of such properties is given in Figure 5.1, where an excerpt of the AADL representation of a MicroBlaze processor is presented. AADL component and property set declarations are used to populate a library of components to be used in FPGA-based designs. Declarations for Xilinx-specific components such as MicroBlaze, buses, controllers, etc. have been added to the library.

5.3.1 Eclipse Modeling Framework

The *Eclipse Modeling Framework* (EMF) [49] [50] is an implementation of the Eclipse IDE [32] dedicated to modeling. EMF thus provides several features:

- A metamodel for class modeling called Ecore, that provides a metamodel which can be used for modeling other models;
- A user interface that allows to display and edit the meta-models;
- Code generation capacities with factories that are able to generate and instantiate Java classes from a model instantiation. It can also perform transformation to others formalism as long as the metamodels and the transformation rules are provided. These transformation rules can be provided in several formalisms such as ATL Transformation Language (ATL) or Query/View/Transformation (QVT).

AADL metamodel relies on the EMF Ecore implementation, which is the generic metamodel used to describe all other metamodels.

5.4 Component Models

We developed component models for each type of component that can be found in embedded systems such as processors, memories, buses, etc. They were described as

```
Package xilinx_components
public
with microBlazeProperties;

  processor microblaze
    features
      reset: in event port;
      interrupt: in data port;
      data_plb: requires bus access;
      inst_plb: requires bus access;
      data_xcl : requires bus access;
          inst_xcl: requires bus access;
      data_lmb: requires bus access;
      inst_lmb: requires bus access;
      debug: requires bus access;
      master_fsl: requires bus access;
      slave_fsl: requires bus access;

    properties
    --default values
      microblazeProperties::FSL_links => 0;
  end microblaze;
end xilinx_components;
```

Figure 5.1: Excerpt of the AADL model of the MicroBlaze.

Java objects with the parameters necessary in order to be used for performances and cost evaluation during DSE along with the necessary element to show their interaction in a design. We also developed more refined AADL models that contain in addition parameters necessary to configure the components in the final design and thus generate the corresponding implementation files.

Since the current framework implementation targets only Xilinx FPGAs, we have developed models for IPs provided with the Xilinx XPS tool (cf. Section 3.2.3 for a description of XPS). This includes the MicroBlaze softcore [48], the memories, the communication and synchronization protocols, the IP to control external peripherals, etc. Each of them were modeled with the necessary parameters so that they could be instantiated in the Xilinx *.mhs* configuration file.

In addition to the descriptions below, all the components are characterized in terms of logic resource costs, in order to be able to estimate the total cost of a system. These values must be provided by the designer. Power consumption models from the Open-PEOPLE project can be added as well. The components can be of four kinds: processing unit, memory, interconnect and external peripheral.

Processing Unit Model

Processing units (PU) are defined in terms of:

- **Type**: MicroBlaze, PowerPC, dedicated IP, etc.
- **Performances**: for each of the task of the application the number of cycles it takes to perform one iteration. These can be automatically filled with the results obtained through the on-target profiling described in Section 3.1.5. In case the PU is a hardware accelerator, then these performances are obtained from the IP characteristics.
- **Memories**: the available memories it can be connected to are defined below. In addition, each memory is characterized by its accessibility: shared memory, local memory, etc.
- **Interconnections**: the available buses the PU and IP can be connected to.

Interconnect Model

Interconnect are characterized with these parameters:

- **Type**: Local Memory Bus (LMB), FSL, Processor Local Bus (PLB), NoC, etc.
- **Bandwidth**: the bandwidth capacity of the link.
- **Latency**: the associated latency of the link.

Memory Model

Memories are modeled with the following characteristics:

- **Type**: DDR, SRAM, BRAM, Flash, etc.
- **Communication interface**: the available communication interfaces to reach the memory (e.g. PLB bus).
- **Bandwidth**: the transfer capacity of the memory in MByte/s.
- **Maximum size**: the maximum size a memory can have. This is necessary for instance to evaluate the cost of synthesized memories such as BRAM.
- **Read & write latency values**: the number of cycles necessary to prepare data before a burst.

External Peripherals

The external peripherals must simply specify what are their requirements in terms of communications: how they can be connected and how much bandwidth they require.

5.5 Specification Template

Architecture templates are used to specify a generic description of the architecture that can be used by our tool as a starting point for the exploration. It also provides the available components for design exploration as well as objectives and constraints to evaluate and bound the design space. This section describes the role and the structure of the template. Figure 5.2 shows an example of an architecture template for a video decoder design. Our template contains three levels of specifications according to the level of expertise or the involvement of designers. The level of specification decides what parts of the design flow our tool deal with, whether it will only take care of the tedious aspects — implementation of the final designs, exploration of the less important system parts, cost and performance evaluations — or if it will have to perform the full design flow with the given minimal constraints.

Static Level The first level, called the *static* level, defines the domain-specific elements of the architecture. These specifications are static parts of the architecture that are specific to the current design and that thus remain constant throughout the DSE. It defines the basic architecture upon which the DSE will rely to perform its exploration. For instance in a video decoder, the bandwidth requirements for data are quite important especially for a decoded raw video stream. Thus some elements of the design are mandatory and are thus decided before the DSE: in the video decoder, it will be a dedicated bus to transfer the decoded video stream to the framebuffer. It is also used to define external peripherals that will be used in the design, such as an output to a screen in a video decoder design. In the Figure 5.2.a), the constant elements are represented with solid lines.

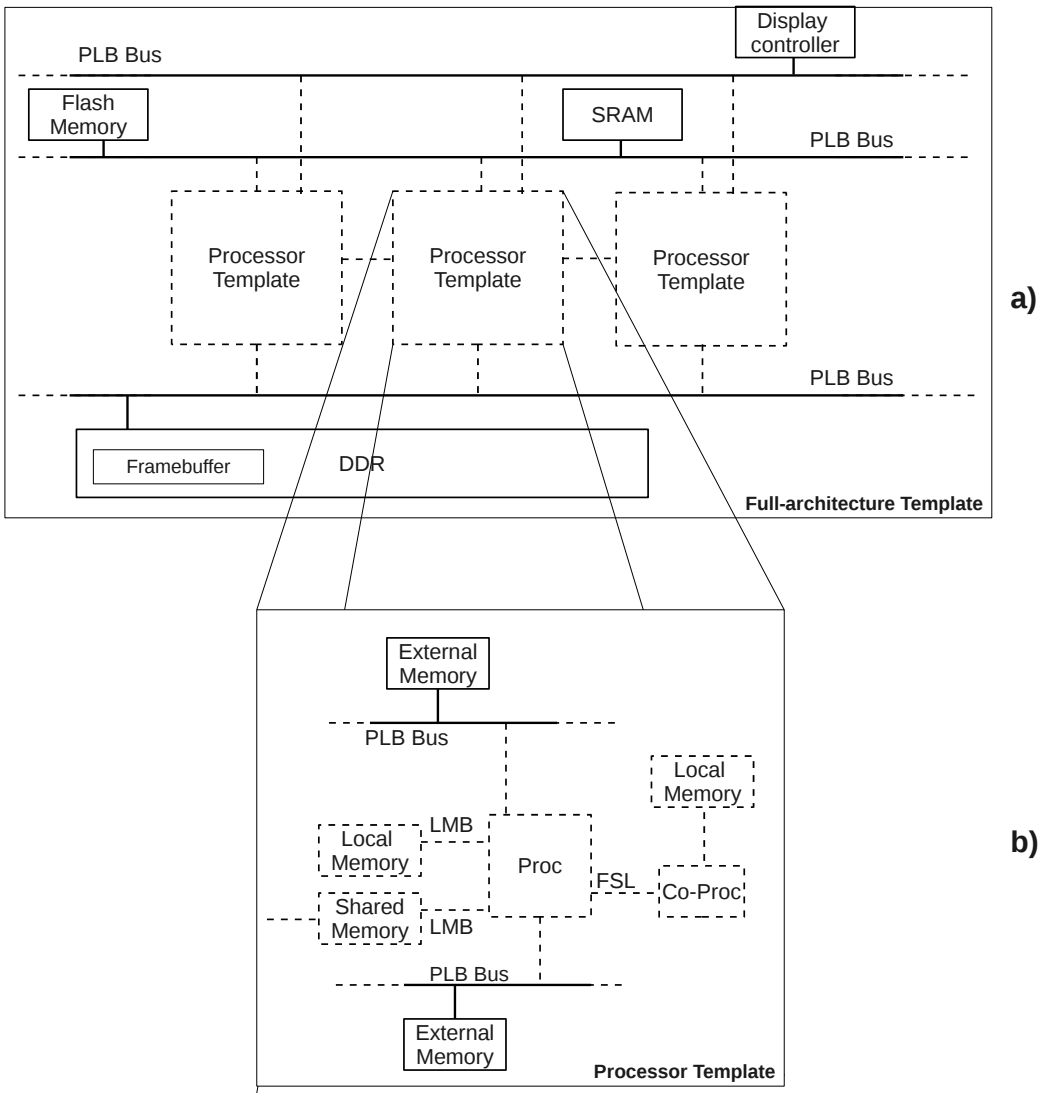

Figure 5.2: Example representation of a template for a video decoder a) of the full architecture; b) detail of the processor template. Solid-lines indicate static elements that remains constant during DSE while dotted-lines represent the possibilities of the architecture according to the constraints set in the DSE-bound level of the template.

DSE-bound Level The second level, called the *DSE-bound* level, specifies the available hardware components, the constraints that define and bound the design space exploration and the performance and cost objective. These specifications are:

- the available types of processors, memories and buses for the DSE;
- the minimum and maximum number of processors in the design;
- the cost constraints in terms of logic resources on the target architecture (e.g. slices, flip-flop, Look-Up Tables (LUT), etc.);
- the performance objective specified in number of cycles. This objective is set for the particular instance of the input application used to evaluate performance with the Sesame tool;
- the specification of the tasks that can be accelerated in hardware and/or that can be duplicated to exploit data-parallelism.

All these specifications define the boundaries of the design space. It is provided to the framework in the form of an XML file.

Expert Level The third level, called the *expert* level, provides to the designer the possibility to specify a priori some of the design decisions that would otherwise be taken by the framework during the DSE. This gives the opportunity to advanced designers to guide the design by relying on their expertise. The possibilities include:

- enforcing the mapping of tasks to specific processing units (GPP or hardware accelerator);
- enforcing the mapping of data on specific memories or channels;
- bypassing the hardware accelerators exploration by specifying the choice of a specific hardware accelerator.

In addition to ensure good design decisions, these enforced decisions have the advantage of reducing the design space. These prunings can then be used to shorten the time of the DSE or to widen the design space by relaxing some of the other constraints. This level is optional, and thus can be ignored by designers less knowledgeable about system design.

Once defined, the templates can be stored in a library so that they can be used as a basis for future designs. By allowing reusability, our template-based approach avoids some tedious design steps and thus relieves designers from repetitive and potentially error-prone tasks.

The template are hierarchical since components themselves be configured following the same decisions: an example for the processor component is given in Figure 5.2.b.

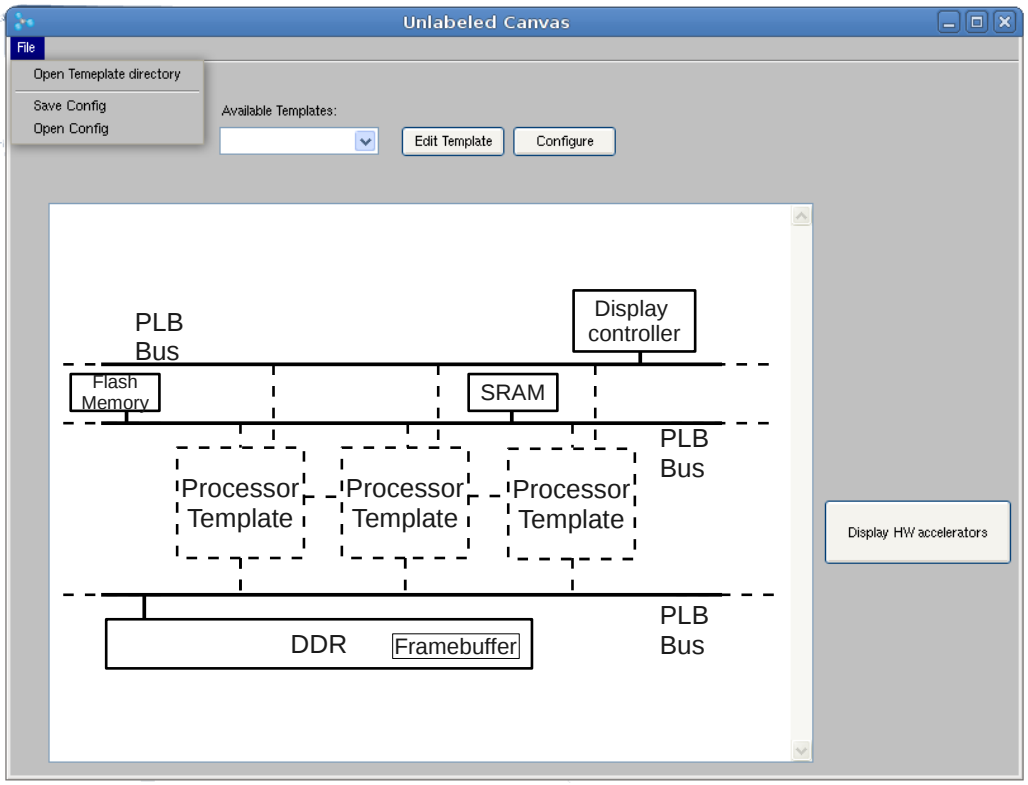

Figure 5.3: Illustration of the template configuration GUI.

5.5.1 Template Configuration Interface

A graphical user interface is under development in order to simplify the template configuration. It shows to the designer a graphical representation of the architecture template. That representation reflects the constraints set by the designer by displaying the current design possibilities: constant elements are shown as solid lines while elements depending on DSE decisions are shown as dotted lines. It allows the designer to define the architecture template graphically by providing a list of the available components that can be instantiated and giving the possibility to place them through drag & drop. It also provides the possibility to make constant some elements of the design. Once the template is defined, the corresponding AADL and XML files are generated to be used as inputs for DSE. An example of what the GUI could look like is shown in Figure 5.3.

5.6 Code Generation

In order to implement the final design selected by the designer at the end of the DSE, we take advantage of the code generation and model transformation capacities offered by the use of MDE.

5.6.1 Software Application Adaptation

In order to adapt the application to the selected architecture design, several modifications have to be made in the C code. These modifications are essentially to adapt the communications accordingly to the data mapping decisions made during the DSE. Depending on the memory where data has to be fetched, it requires to insert the appropriate API parametrized with the appropriate memory address. If a task is implemented as an hardware accelerator, then it is necessary to add to the source code of the tasks communicating with the IP the API implementing the communication protocol used by the bus connecting the IP.

5.6.2 Implementation Project Files

The developed model is used to generate the implementation files for the FPGA backend tools (Xilinx XPS, Altera Quartus). Since we target Xilinx tools, the generated files are:

- The *Microprocessor Hardware Specifications* (.mhs), which describes all the hardware elements used in the design as well as their parameters. An example of instantiation of a MicroBlaze is shown in Figure 5.5.
- The *Microprocessor Software Specifications* (.mss), which describes the drivers for controlling the components from software.

So each component has an AADL model with the properties found in the *.mhs* and *.mss* files of Xilinx projects. In the AADL model, these properties are filled with default values, and the properties which need to be changed to reflect the characteristics of the

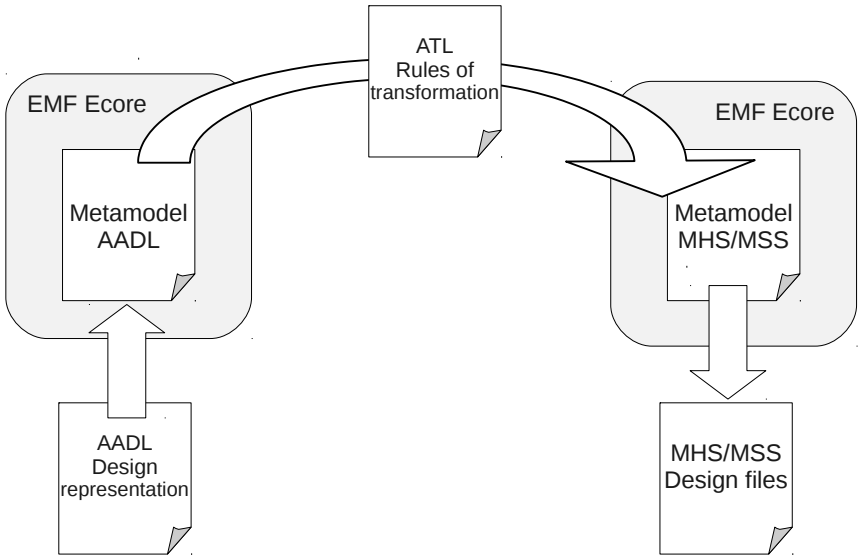

Figure 5.4: The model transformation process.

generated design are updated at the end of the DSE. In the Xilinx *.mhs* formalism [51], the instantiated components are described through three kind of characteristics:

- **Bus interface** which specifies the connecting bus protocols used by the component and the instance of the bus to which it is connected.
- **Port** which specifies the component connection ports and the system port to which it is assigned.
- **Parameter** which specifies the options to configure the component (e.g. the size of a MicroBlaze data cache).

Figure 5.5 shows the instanciation of a MicroBlaze in an *.mhs* file.

From the AADL description of the selected architecture, it is possible to perform a model transformation to generate the *.mhs* and *.mss* implementation files. For this, metamodels for both the input and output models are needed, and a set of transformation rules — for example in ATL— must be provided. The transformation process is illustrated in Figure 5.4. The result is a nearly-ready-to-implement project, that can be opened and edited with Xilinx XPS and which can be used to synthesize and implement the project on the target FPGA.

5.7 Conclusion

In this chapter, a template-based approach for design of H-MPSoC was introduced. This template is used to describe the generic architecture used as basis for the DSE. Its multiple levels of specifications allows designers to provide their specifications according

```
BEGIN microblaze
 PARAMETER INSTANCE = microblaze_1
 PARAMETER C_USE_FPU = 2
 PARAMETER C_DEBUG_ENABLED = 1
 PARAMETER C_ICACHE_BASEADDR = 0x90000000
 PARAMETER C_ICACHE_HIGHADDR = 0x9fffffff
 PARAMETER C_CACHE_BYTE_SIZE = 8192
 PARAMETER C_ICACHE_ALWAYS_USED = 1
 PARAMETER C_DCACHE_BASEADDR = 0x90000000
 PARAMETER C_DCACHE_HIGHADDR = 0x9fffffff
 PARAMETER C_DCACHE_BYTE_SIZE = 8192
 PARAMETER C_DCACHE_ALWAYS_USED = 1
 PARAMETER HW_VER = 7.30.b
 PARAMETER C_USE_ICACHE = 1
 PARAMETER C_USE_DCACHE = 1
 PARAMETER C_USE_BARREL = 1
 PARAMETER C_USE_DIV = 1
 PARAMETER C_NUMBER_OF_PC_BRK = 1
 PARAMETER C_NUMBER_OF_RD_ADDR_BRK = 0
 PARAMETER C_NUMBER_OF_WR_ADDR_BRK = 0
 BUS_INTERFACE DPLB = mb_plb_1
 BUS_INTERFACE IPLB = mb_plb_1
 BUS_INTERFACE DXCL = microblaze_1_DXCL
 BUS_INTERFACE DEBUG = microblaze_1_mdm_bus
 BUS_INTERFACE IXCL = microblaze_1_IXCL
 BUS_INTERFACE DLMB = dlmb_1
 BUS_INTERFACE ILMB = ilmb_1
 PORT MB_RESET = mb_reset
 PORT INTERRUPT = microblaze_1_Interrupt
END
```

Figure 5.5: Example of the instantiation of a MicroBlaze in a Xilinx MHS files. It specifies the instantiation options, the bus and port connections.

to their level of expertise. This template, coupled with MDE methods favors design reuse and portability thus reducing the cost of H-MPSoC designs. The use of the AADL DSL to describe the specifications eases the conception by abstracting many implementation details and also enables model transformation and code generation in order to automatize the implementation of the final design.

6 Results

The framework has been tested with two real applications: an MJPEG decoder and a face-detection application. These applications are used to check the accuracy of the estimations, the performance of the framework and the quality of the final produced results. These applications have different characteristics and consequently each was used to validate different parts of the exploration. In addition, we also run several benchmarks in order to validate more specific points of the framework. All the results presented in this chapter were performed on a desktop computer equipped with a quadcore Intel Xeon W3520 @ 2.67GHz, 8 Gb of RAM and the 64-bit version of Debian Squeeze operating system.

6.1 Application 1: MJPEG decoder

The first application is a Motion JPEG decoder [52]. Motion JPEG is a video format that encodes video streams as a succession of pictures independently encoded in the JPEG format.

6.1.1 Presentation

The decoding of an MJPEG encoded video stream consists in performing a series of operations:

- It first starts by decoding the picture by analyzing the elements of the video stream, seeking for specific markers which provide information such as the end of a frame, the size of the frame, etc.
- Then the Huffman decoding is performed. Huffman coding consists in reducing the size of the data by encoding the most frequent elements with shorter symbols, i.e. the most common elements will take less place in memory. Huffman decoding is thus the inverse operation.
- Next the Inverse Discrete Cosine Transform (IDCT) transforms back the encoded data into the spatial domain.
- Finally, the colors which were coded with chrominance and luminance (YUV) are converted back to red, green and blue color-values (RGB). This is done through a

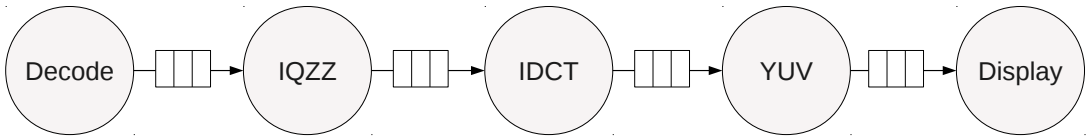

Figure 6.1: Representation of the Kahn Process Network of the implementation of the MJPEG decoder split in 5 tasks.

series of three vector multiplications (one by color). The result is a frame of pixels each encoded on 24 bits (8 bits per color) that can be copied into a framebuffer for display.

6.1.2 Specifications

The implementation that was made for the case study consists of five C code tasks, as illustrated by the graph in Figure 6.1:

- *Decode* is the task that reads data from the file and performs the video decoding as well as the Huffman decoding. The output is
- *IQZZ* is the task that performs the inverse quantization.
- *IDCT* is the task that performs the IDCT.
- *YUV* is the task that converts the colors from the YUV format to the RGB format.
- *Display* which copies the decoded video data in the framebuffer.

The movie used as example for the implementation is a video in VGA format, i.e. frames have a resolution of 256 $\times$ 144 pixels. The target for the design is the Xilinx XUPV5 FPGA board [53].

6.1.3 Results

We will now present the results by following the design flow step by step.

Application Parallelization

First a profiling of the application is performed in order to guide designers in the splitting of the application into tasks. They must specified the application in the form of a SANLP, so that it can then be automatically transformed into KPN formalism. This is performed by the couple of tools KPNGen and ESPAM: the former transforms the application graph while the latter generates the corresponding C code.

Table 6.1: Results of the profiling on target.

Task	**Percentage**	**Cycles count**	**Number of cycles for one iteration**
Decoding	19.4 %	1297871068	242140
Quantification	9.3 %	623876806	116395
IDCT	47.2 %	3152960456	588335
YUV	18.4 %	1221882386	227963
Display	5.7 %	377403958	70411

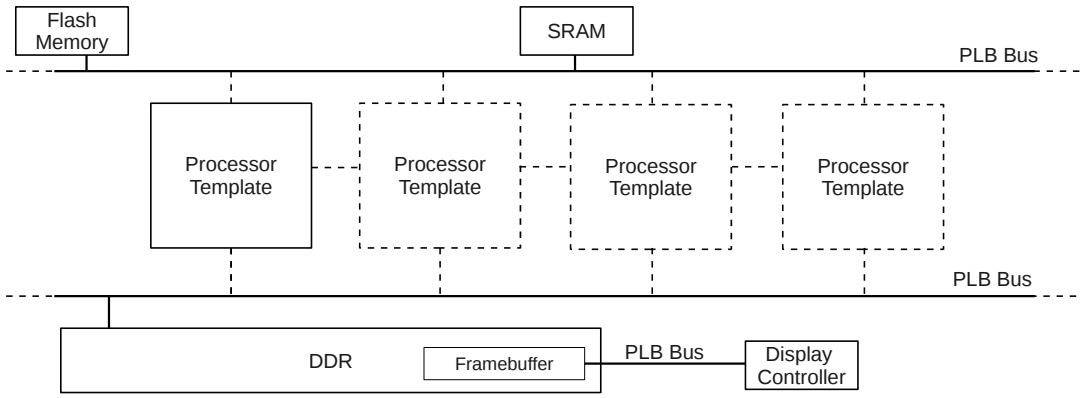

Figure 6.2: Architecture template for the MJPEG decoder

On-Target Profiling

The transformed code then undergoes a profiling on-target, which is a MicroBlaze [48] in this case. The code is adapted for MicroBlaze by a modified version we made of the ESPAM code generator. Profiling results are shown in Table 6.1. With these values, we can sort the tasks according to their computation resource requirements and we can compute the number of cycles necessary for one iteration of the task which is a requirement for performance estimations later in the DSE.

DSE Settings

The settings for the DSE were given in the template. The generic architecture is shown in Figure 6.2. In this figure, elements specific to this application are solid lines: so the architecture will contain a Flash memory that will be used to provide the input video file, a SRAM memory — both memories are linked to the processor through a PLB bus — and a framebuffer implemented in the DDR memory that is linked through a dedicated PLB bus to the display controller. The dotted lines elements show components that can be instantiated. So we can see here that there can be up to four

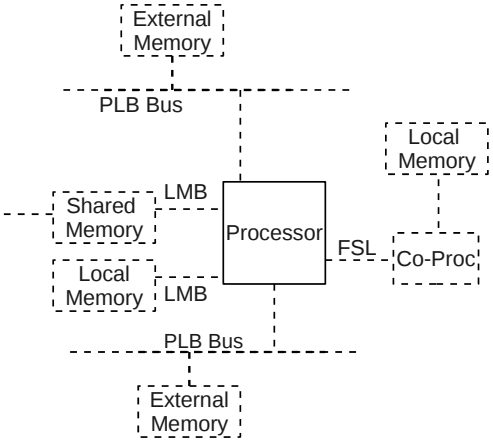

Figure 6.3: Processor template for the MJPEG decoder

processors instantiated. Figure 6.3 shows the detail of the processor template. It shows that processors can be linked to a local memory or a shared memory through a LMB bus. It can also be linked to one or more external memories through a PLB bus and can also be linked through a FSL to a coprocessor which can itself has a dedicated local memory. These architectural elements represent the static level of the template. The rest of the settings for the DSE-bound level are:

- The number of processors is between one and four and all of them are of MicroBlaze type.
- Tasks IDCT and YUV can be accelerated through hardware.
- The IDCT can also be duplicated to exploit data-parallelism.
- The target board is a Xilinx XUPV5 FPGA board [53].
- Possible memory implementations were: DDR, BRAM and SRAM.
- The performance objective is to reach 24 frames per second (FPS).
- The N_X variables that set the scalability of the DSE were all set to -1, meaning that no pruning was performed.

No expert level constraint is specified in this case.

DSE Results

With these inputs, the actual DSE begins. During DSE, exploration is performed for hardware accelerators for the IDCT task. The results of the exploration are given in Section 6.2.1.

The output of the DSE is a set of Pareto Optimal solutions with the estimated performance obtained through the Sesame simulation tool and the estimation of logic resources cost. The results are given in Table 6.2 and in Figure 6.4.

In Table 6.2, the Architecture column is the description of the architecture: MB YUV IDCT_X means an architecture with three processors where MB means MicroBlaze, YUV is a MicroBlaze with YUV hardware accelerators and IDCT_X is a MicroBlaze with one the version IDCT hardware accelerator — the lower the number X, the faster the IP, according to the HLS-based exploration. The second column specifies the task mapping: there as many numbers as there are tasks, and each number represents a processing unit, in the same order as they are described in the first column. For instance, for the architecture MB IDCT_5 YUV, MB is the processor 0, IDCT_2 is the processor 1 and YUV is the processor 3. Thus the mapping 0 1 1 2 2, means that the first task (*Decode*) is mapped on MB, the second and third tasks (*IQZZ* and *IDCT*) are mapped on IDCT_5 and that the last two tasks (*Display* and *YUV*) are mapped on YUV. The next column, Sesame estimation, is the number of cycles estimated by Sesame for the execution of the application. FPS is the number of decoded Frames per Second, based on the assumption that the architecture will run at 125 MHz. The last three columns shows the logic resource cost estimations, in terms of *Slices*, *BRAM* and *DSP*.

If we excludes the time for the hardware accelerators exploration — by supposing for instance that the IPs were already present in the database — , the total time taken to produce these results was of 21 seconds, about 13 of which were taken by the Sesame trace-based performance simulation. We can see that seven architectures were found that reach our objective of 24 FPS.

Hungarian Method Efficiency Evaluation

In order to evaluate the efficiency of the Hungarian algorithm method (cf. Section 4.6.2), we compared the best mappings found with the Hungarian method and the optimal mappings found with an exhaustive mapping exploration. The results are given in Table 6.3 where they are sorted in increasing order of the Sesame estimation of the exhaustive mappings. We can see that in 11 cases out of 37, there are no or very little difference in performances, meaning that our Hungarian method has found the optimal solution or a near-optimal one. In average the difference in performances is 12%. This difference is relatively small especially given that, if the mapping generation was exhaustive, then over 200 000 architectures and mappings would have been generated which would have taken over a day to evaluate with the Sesame tool. This is to be put against the 37 architectures and mappings generated with the Hungarian method, which were evaluated in about 13 seconds. This corresponds to a speedup over 6600, meaning that a loss of one percent of performance, lead to a speedup of 550. Our method thus provides a huge speedup over an exhaustive exploration for the mapping as performed in the Daedalus framework [4].

Moreover the Hungarian method find a solution in polynomial time, which means that for larger applications — i.e. with more tasks — or larger architectures — i.e.

Table 6.2: Results of the exploration by our framework for the MJPEG decoder.

Architecture	**Mapping**	**Sesame Est.**	**FPS**	**Slices**	**BRAM**	**DSP**
MB	0 0 0 0 0	718915412	5	2103	0	0
MB MB	0 1 1 0 0	422931756	8	4206	0	0
MB YUV	0 0 1 1 0	393331640	9	2643	1	4
MB MB MB	2 2 1 0 0	334121151	10	6309	0	0
MB MB YUV	1 1 0 2 2	333971982	10	4746	1	4
MB MB MB MB	3 0 2 1 1	333798895	10	8412	0	0
MB MB MB YUV	1 0 2 3 0	333650802	10	6849	1	4
MB IDCT_5	1 1 1 0 0	276949605	13	8286	8	6
MB IDCT_4	1 1 1 0 0	273862245	13	9519	9	9
MB IDCT_3	1 1 1 0 0	272704485	13	9913	9	9
MB IDCT_2	1 1 1 0 0	270388965	13	10525	10	12
MB IDCT_1	1 1 1 0 0	269231205	13	11931	11	15
MB IDCT_0	1 1 1 0 0	268459365	13	12573	11	18
MB MB IDCT_5	2 1 2 0 0	209843700	17	10389	8	6
MB MB IDCT_0 YUV	3 0 2 3 0	203276335	17	15216	12	22
MB MB IDCT_1 YUV	3 0 2 3 0	203276335	17	14574	12	19
MB MB IDCT_2 YUV	3 0 2 3 0	203276335	17	13168	11	16
MB IDCT_3 YUV	2 1 1 2 0	203274183	17	10453	10	13
MB IDCT_4 YUV	1 0 1 2 0	183189421	19	10059	10	13
MB MB MB IDCT_5	1 2 3 3 2	176360236	20	12492	8	6
MB MB MB IDCT_4	1 2 3 3 2	172228403	21	13725	9	9
MB MB IDCT_4	1 0 2 2 0	172226236	21	11622	9	9
MB MB MB IDCT_3	1 2 3 3 2	170678956	21	14119	9	9
MB MB IDCT_3	1 0 2 2 0	170676796	21	12016	9	9
MB MB MB IDCT_2	1 2 3 3 2	167580076	21	14731	10	12
MB MB IDCT_2	1 0 2 2 0	167577916	21	12628	10	12
MB MB MB IDCT_1	1 2 3 3 2	166030636	21	16137	11	15
MB MB IDCT_1	1 0 2 2 0	166028476	21	14034	11	15
MB MB MB IDCT_0	1 2 3 3 2	164997676	21	16779	11	18
MB MB IDCT_0	1 0 2 2 0	164995516	21	14676	11	18
MB MB IDCT_5 YUV	0 2 2 3 1	144090410	25	10929	9	10
MB IDCT_5 YUV	0 1 1 2 2	144090406	25	8826	9	10
MB MB IDCT_4 YUV	1 2 2 3 0	144083804	25	12162	10	13
MB MB IDCT_3 YUV	0 2 2 3 1	144079850	25	12556	10	13
MB IDCT_2 YUV	0 1 1 2 2	144074086	25	11065	11	16
MB IDCT_1 YUV	0 1 1 2 2	144071206	25	12471	12	19
MB IDCT_0 YUV	0 1 1 2 2	144069286	25	13113	12	22

with more processing units —, the computation time will remain reasonable.

Comparison with FPGA Implementations

Finally we implemented the design on the FPGA in order to check the accuracy of our estimations. The results are given in Table 6.4. Architectures with more processors could not be implemented, as they do not fit on the XUPV5 board. These architectures were not rejected during the DSE, due to the fact that our logic resource estimation does not take into account the cost due to placing and routing. That could be solved in future version by specifying a percentage of the FPGA resources that can be used, in order to let a safety margin for placing and routing. A threshold of 80% is for instance a reasonable value based on experiments.

The average error is 7.32% which is quite reasonable considering the speedup obtained over the implementation time on FPGA. While our evaluations took a couple of seconds to be computed, the implementation of the simplest architecture — i.e. monoprocessor — took around half an hour.

Conclusion

Through this case-study we have validated our exploration algorithm, including our fast task-mapping method. We have also shown the accuracy of our performance estimations by comparing them with real on-target implementations.

6.2 Hardware Accelerators Exploration

In this section we evaluate the accuracy of our logic resource estimation as well as the speedup obtained over logic synthesis-based estimation.

6.2.1 IDCT IP Exploration

During the exploration for the MJPEG decoder application, two tasks were tagged as candidates to hardware acceleration: IDCT and YUV. As a result, series of the corresponding IPs were generated by our tool, following the process described in Section 4.3. The chosen model of communication was FIFO channels implemented in BRAM. Figure 6.5 shows the results for the series of IPs generated for the IDCT. We can see that our estimations of the consumed resources are quite accurate for the slices since the actual number of consumed slices obtained after logic synthesis is bounded by our upper and lower estimations. The BRAM consumption is almost accurately predicted in all cases with only one value not correctly estimated. The vertical red lines show the six Pareto optimal solutions that are kept for the DSE. These IPs are presented in detail in Table 6.5. The exploration of these 56 IPs took about five minutes. This time includes the cost estimations which are almost instantaneous since it consists in a set of simple arithmetic and logic rules, most of the time being consumed by the HLS.

Table 6.3: Comparison between the mappings found with the Hungarian algorithm method and the exhaustive mapping exploration.

	Hungarian Method		**Exhaustive Mappings**		**Difference**	**Difference %**
Architecture	**Mapping**	**Sesame Est.**	**Mapping**	**Sesame Est.**		
MB	0 0 0 0 0	718915412	0 0 0 0 0	718915412		
MB MB	0 1 1 0 0	422931756	0 0 1 0 1	402288504	20643252	4,88
MB YUV	0 0 1 1 0	393331640	1 1 0 1 0	347565402	45766238	11,64
MB MB MB MB	3 0 2 1 1	333798895	0 1 2 3 0	333798893	2	0
MB MB MB	2 2 1 0 0	334121151	0 1 2 0 0	333798887	322264	0,1
MB MB MB YUV	1 0 2 3 0	333650802	3 0 1 3 2	333649734	1068	0
MB MB YUV	1 1 0 2 2	333971982	2 0 1 2 0	333649726	322256	0,1
MB IDCT_5	1 1 1 0 0	276949605	1 0 1 0 1	229034147	47915458	17,3
MB IDCT_4	1 1 1 0 0	273862245	1 0 1 0 1	229026467	44835778	16,37
MB IDCT_3	1 1 1 0 0	272704485	1 0 1 0 1	229023587	43680898	16,02
MB IDCT_2	1 1 1 0 0	270388965	1 0 1 0 1	229017827	41371138	15,3
MB IDCT_1	1 1 1 0 0	269231205	1 0 1 0 1	229014947	40216258	14,94
MB IDCT_0	1 1 1 0 0	268459365	1 0 1 0 1	229013027	39446338	14,69
MB MB IDCT_5	2 1 2 0 0	209843700	0 2 2 1 2	144263085	65580615	31,25
MB MB IDCT_4	1 0 2 2 0	172226236	0 2 2 1 2	144255405	27970831	16,24
MB MB IDCT_3	1 0 2 2 0	170676796	0 2 2 1 2	144252525	26424271	15,48
MB MB IDCT_2	1 0 2 2 0	167577916	0 2 2 1 2	144246765	23331151	13,92
MB MB IDCT_1	1 0 2 2 0	166028476	0 2 2 1 2	144243885	21784591	13,12
MB MB IDCT_0	1 0 2 2 0	164995516	0 2 2 1 2	144241965	20753551	12,58
MB MB MB IDCT_5	1 2 3 3 2	176360236	0 1 3 2 1	144238499	32121737	18,21
MB MB MB IDCT_4	1 2 3 3 2	172228403	0 1 3 2 1	144230819	27997584	16,26
MB MB MB IDCT_3	1 2 3 3 2	170678956	0 1 3 2 1	144227939	26451017	15,5
MB MB MB IDCT_2	1 2 3 3 2	167580076	0 1 3 2 1	144222179	23357897	13,94
MB MB MB IDCT_1	1 2 3 3 2	166030636	0 1 3 2 1	144219299	21811337	13,14
MB MB MB IDCT_0	1 2 3 3 2	164997676	0 1 3 2 1	144217379	20780297	12,59
MB MB IDCT_5 YUV	0 2 2 3 1	144090410	0 2 2 3 1	144090410	0	0
MB IDCT_5 YUV	0 1 1 2 2	144090406	0 1 1 2 2	144090406	0	0
MB MB IDCT_4 YUV	1 2 2 3 0	144083804	0 2 2 3 1	144082730	1074	0
MB IDCT_4 YUV	1 0 1 2 0	183189421	0 1 1 2 2	144082726	39106695	21,35
MB MB IDCT_3 YUV	0 2 2 3 1	144079850	0 2 2 3 1	144079850	0	0
MB IDCT_3 YUV	2 1 1 2 0	203274183	0 1 1 2 2	144079846	59194337	29,12
MB MB IDCT_2 YUV	3 0 2 3 0	203276335	0 2 2 3 1	144074090	59202245	29,12
MB IDCT_2 YUV	0 1 1 2 2	144074086	0 1 1 2 2	144074086	0	0
MB MB IDCT_1 YUV	3 0 2 3 0	203276335	0 2 2 3 1	144071210	59205125	29,13
MB IDCT_1 YUV	0 1 1 2 2	144071206	0 1 1 2 2	144071206	0	0
MB MB IDCT_0 YUV	3 0 2 3 0	203276335	0 2 2 3 1	144069290	59207045	29,13
MB IDCT_0 YUV	0 1 1 2 2	144069286	0 1 1 2 2	144069286	0	0

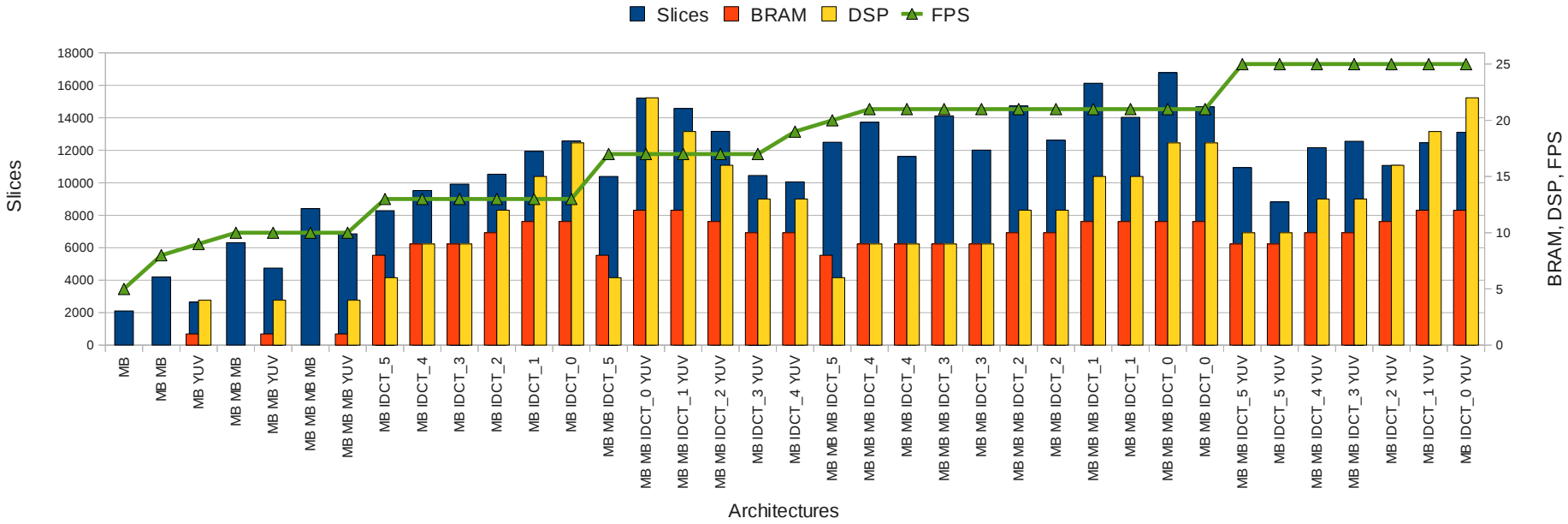

Figure 6.4: Results of the exploration of the MJPEG by our framework sorted by increasing FPS.

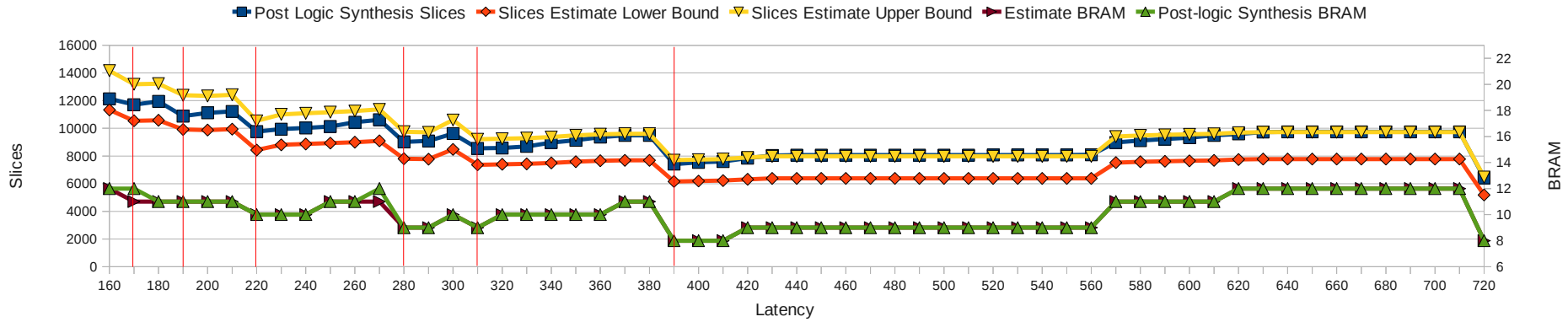

Figure 6.5: Comparison of the number of consumed slices and BRAM in our estimation and the actual number obtained after logic synthesis. The vertical lines indicate the IDCT IPs corresponding to Pareto points which are the IPs that were finally selected.

Table 6.4: Comparison of the implementations on FPGA of different MJPEG architectures and the performance evaluations given by our tool.

Architecture	**FPS**	**Est. FPS**	**Cycles count**	**Est. Cycles count**	**% Error**
Monoprocessor – Full Software	5.32	5	681059750	718915412	5.56
Monoprocessor – Hardware YUV	5.71	5	634948450	639242454	0.7
Monoprocessor – Hardware IDCT_0	9.48	8	382378299	412204302	7.8
Dual Processor – Full Software	9.95	8	364346144	422931756	16.08
Dual Processor – Hardware YUV	10.43	9	347540716	393331640	13.18
Dual Processor – Hardware IDCT_0	13.42	13	270104971	268459365	0.61

Table 6.5: Characteristics of the chosen IDCT IPs: latency, post logic synthesis resources cost, the error in our estimation of the slices cost and the time speedup between our estimation and logic synthesis

IP Version	IDCT1	IDCT2	IDCT3	IDCT4	IDCT5	IDCT6
Latency	170	190	220	280	310	390
Slices	11707	11115	9767	9019	8551	7410
Slices Cost Error	3,9%	2,36%	4,18%	3,17%	2,73%	2,52%
BRAM	12	11	10	9	9	8
Speedup	50	57	44	41	64	43

6.2.2 Benchmark

In order to validate the accuracy of our cost estimation, we launched series of hardware exploration for ten functions: FFT, IDCT, IDCT2D, IIR filter, FIR filter, CORDIC (arcos), Sobel filter, Gaussian filter, Walsh-Hadamard transform and YUV conversion. For each of these functions, several implementations were considered: with communications whether in FIFO and as Ping-Pong memory and with the FSM whether implemented in BRAM or in LUT. Results are given in Table 6.6 in the form of triplets: minimum, average and maximum values respectively. For all the implementations the average error is at most of 10% which proves that our estimation technique is pretty accurate. The speedup over logic synthesis is on average of two orders of magnitude.

The speed and the accuracy of our technique shown by these results prove that it is possible to integrate the exploration of hardware accelerators as a new step of the DSE loop of a MPSoC.

6.3 Application 2: Face detection with the Viola-Jones algorithm

As a second benchmark for our framework, we used a face-detection application based on the Viola-Jones algorithm [54]. This application is directly inspired from the implementation available under free license at OpenCV [55].

Table 6.6: Accuracy of cost estimation and measured accuracy for series of IPs generated for 10 different functions.

[min, avg, max] errors	**DSP**	**REG (%)**	**LUT (%)**	**Slices (%)**	**BRAM (%)**	**Speedup (XST/HLS)**
LUT/FIFO	0	[0,0.5,5]	[5,8,16]	[2,10,16]	—	[82,150,556]
LUT/Ping-Pong	0	[1,5,9]	[1,8,10]	[2,8,19]	—	[20,120,756]
BRAM/FIFO	0	<1	[1,8,16]	[4,9,17]	[0,∼0,100]	[50,150,520]
BRAM/Ping-Pong	0	[2,4,10]	[1,7,19]	[1,10,19]	[0, 2, 10]	[51,110,1160]

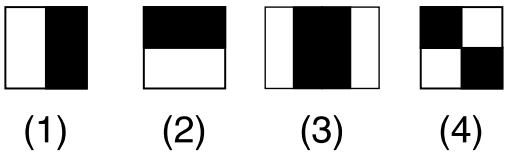

Figure 6.6: Examples of Haar-like features.

6.3.1 Presentation

The face-detection is based on the Viola-Jones techniques for object-recognition in pictures. This technique is based on the computation of patterns in an image to check the presence of characteristics of the object that needs to be detected. These characteristics are computed using so-called Haar-like features, which are simple rectangular patterns that are used to characterized parts of a pictures. An example of the patterns used as Haar-like features is given in Figure 6.6. So to detect a specific object, it is necessary to train a set of classifiers for this specific object, e.g. a human face. This results in a set of classifiers that are efficient to detect the object (i.e. that yields few false negatives) and to reject parts of the picture that do not contains the object (i.e. that yields few false positives). In our case, the learning algorithm used to train the classifiers for human face detection was AdaBoost [56].

Haar-like features are made of patterns containing between two to four rectangles of the image pixels. For instance in Figure 6.6, the value of a feature is computed by subtracting the sums of the pixels of the white rectangle(s) to the sum of the pixels in the black rectangle(s). In order to speedup the computation of the Haar-like features, we used the *integral image* method, which accelerates the computation of rectangular areas of an image.

The integral image consists in calculating for each pixel of the image, the sum of all the pixels which have their (x, y) coordinates smaller than the current pixel. This sum represents the value of the rectangular area formed by the origin of the image and the pixel, as illustrated by Figure 6.7.a. This accelerates greatly the computation of Haar-like features since it enables the computation of the sum of a rectangle of pixels to be performed in constant time. Once the integral image computed, the sum of a rectangle of pixels for which the corners would be the pixels A, B, C, and D can be computed with the formula: $((II_D - II_B) - II_C) + II_A$, where II_K is the integral image

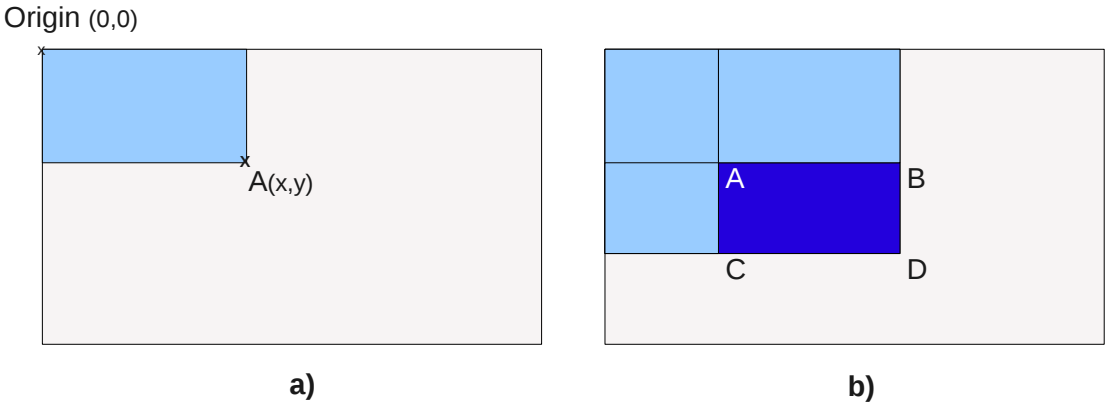

Figure 6.7: a) In integral image, the value of the pixel A is the sum of all the pixels in the colored rectangle. b) The computation of the rectangle defined by the pixels $ABCD$ is given by the formula $((D - B) - C) + A$.

value of the pixel K. The computation of a rectangular value, illustrated by Figure 6.7.b, thus consists in one addition and two subtractions.

The face detection itself is then performed on the picture, by scanning it with a search window specifying the area where face detection is performed. This window is moved in order to performed the detection on every part of the image. For each zone specified by the window, the computation of the Haar-like features is performed. In order to optimize the process, the Haar-like features are split into stages, the first stages being less computation-intensive and discarding more quickly a zone where there is no face. So if at a stage the detection is negative, the application moves to the next zone of the picture, otherwise it goes to the next stage of classifiers. If for a zone, no classifier provides a negative result, then it is assumed that the zone contains a face. Once the whole picture has been scanned, the search window is resized according to a scaling factor, in order to perform the search at a higher scale. The scaling goes on until the window becomes bigger than the picture. The final result is a set of coordinates of areas containing faces. Since a face might have been detected multiple times, it might be necessary to merge the overlapping areas so that each face is detected only once.

6.3.2 Specifications

In our implementation of the face-detection algorithm, we first perform an initialization phase to process the image in order to optimize and accelerate detections. This initial image processing is used to enhance the image so that the detection will be faster and yield better results. The processing operations we perform are:

- **converting the image from colors to shades of gray**: this reduces the size of the processed data since a color pixel is usually encoded on 24 bits (8 bits per color — red, green and blue) while a gray pixel is encoded on 8 bits.
- **reducing the size of the image by four**, from 640×480 to 320×240: this

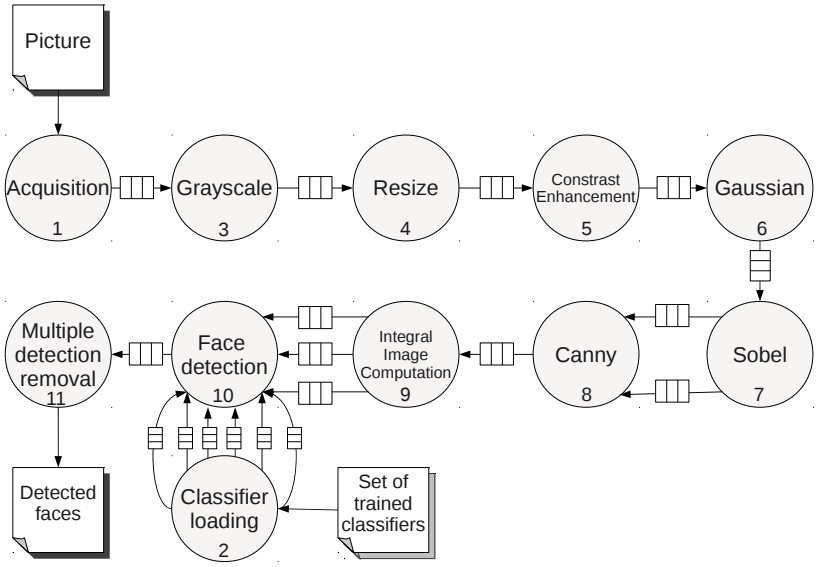

Figure 6.8: The face detection application split in eleven tasks with the different links between communicating tasks.

operation also reduces the size of data to work on without degrading the results on images where faces are not too small.

- **enhancing the contrast**: this is done through the histogram equalization technique which stretches the values of the pixel in shades of gray to the full range of possible values on 8 bits (i.e. from 0 to 255).
- **applying a Gaussian blur**: this blur is a preliminary operation that allows a better edge detection.
- **Sobel processing**: this is an edge detection method used to prune the area where face detection is applied: parts of the picture that contain no edge are very likely to contain no face, so detecting these areas allows to avoid to waste time computing classifiers for these areas.
- **Canny edge detection**: a technique used after Sobel to put even more in evidence the edges in the picture.

All these steps are illustrated in Figure 6.9. Once these treatments are done, the integral image is computed and the actual face detection phase is launched.

6.3.3 Results

Our implementation of the face-detection application is composed of 11 tasks as illustrated by Figure 6.8. There is one task for each of the six operations described above (*Grayscale, Resize, Contrast enhancement, Gaussian, Sobel, Canny*); two tasks

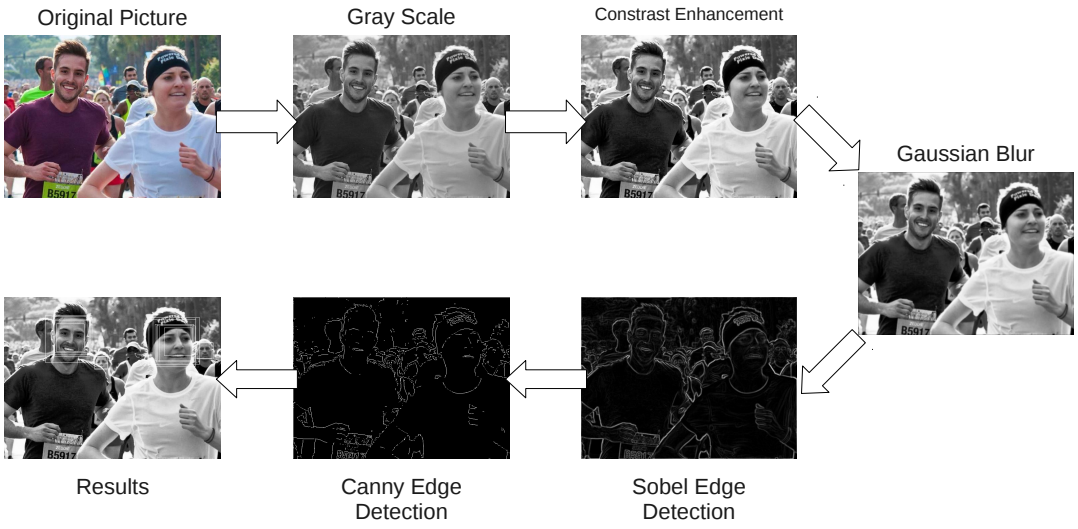

Figure 6.9: Illustration of the operations performed on the picture and of the final result showing the detected faces.

for the acquisition: one for the image data, one for the classifiers (*Image acquisition* and *Classifier loading*); one task for the computation of the integral image (*Integral image computation*), one task for the actual face detection (*Face detection*) and finally one task for the removal of the multiple detections (*Multiple detection removal*).

Since the whole set of classifiers is quite large and since the first stages are called more often than the others — and are also smaller in size —, we split the classifier into six sets of stages. The splitting is made so that the first four stages have their own communication channel, and the rest of the 25 stages are split in two, each having their own communication channel. This splitting allows us to have a more refined control over the data mapping. The different types of data and their sizes are given in Table 6.7.

On-Target Profiling

Similarly to what was done for the MJPEG decoder, the application is parallelized with KPNGen, before undergoing the on-target profiling. The results of the profiling are shown in Table 6.8. We can see from this profiling that three tasks consume 90% of the execution time: *Face Detection*, *Classifier Loading* and *Contrast Enhancement*. Efforts for optimization should consequently be focused on those tasks. In this example, each task is called only once, except for the task performing the face detection, which iterates as many times as the number of scales of the searching window.

Table 6.7: List of the communication channels between tasks with the type and size of data exchanged

Link	**Type**	**Size (bytes)**
1-3	IMG_COLOR	921600
2-10	CLASSIFIER_1	3504
2-10	CLASSIFIER_2	6220
2-10	CLASSIFIER_3	10848
2-10	CLASSIFIER_4	13328
2-10	CLASSIFIER_5_12	261012
2-10	CLASSIFIER_13_25	935892
3-4	IMG_GRAY	311964
4-5	IMG_RESIZE_GRAY	65416
5-6	IMG_RESIZE_GRAY	65416
6-7	IMG_RESIZE_GRAY	65416
7-8	IMG_RESIZE_GRAY	65416
7-8	IMG_RESIZE_GRAY	65416
8-9	IMG_RESIZE_GRAY	65416
9-10	IMG_INTEGRAL	279056
9-10	IMG_INTEGRAL	279056
9-10	SQUAREIMG_INTEGRAL	558080
10-11	RESULTS	2400

Table 6.8: Results of the profiling on target for the face detection algorithm.

Task	**Percentage**	**Cycles count**
Image acquisition	7.32 %	174116541
Classifier loading	18.29 %	435077247
Grayscale	0.29 %	6908604
Resize	1.71 %	40583696
Contrast enhancement	9.09 %	216229518
Gaussian	1.85 %	44089594
Sobel	0.71 %	16768814
Canny	0.57 %	13565736
Integral image computation	0.38 %	9065774
Face detection	59.68 %	1419321643
Multiple detection removal	0.1 %	2496220

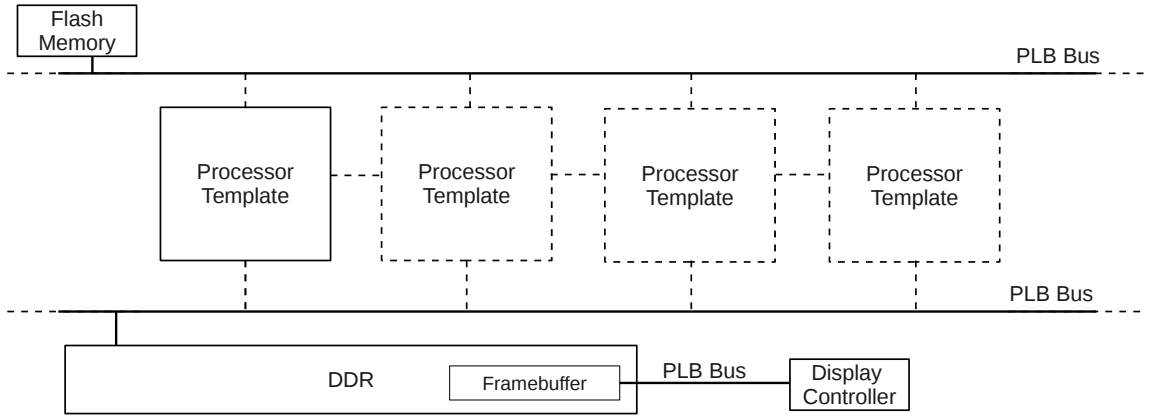

Figure 6.10: Architecture template for the face detection application

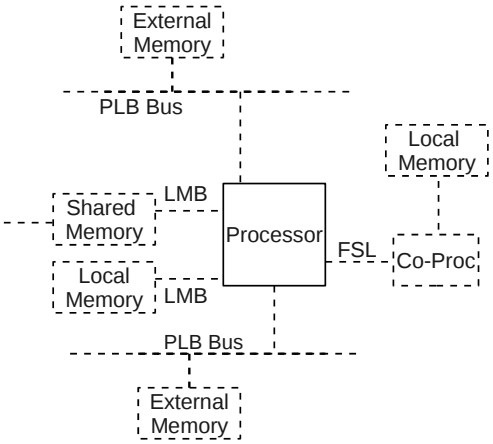

Figure 6.11: Processor template for the MJPEG decoder

DSE Settings

Figure 6.10 shows the generic architecture for the face detection application. It is very similar to the one for the MJPEG decoder, however there is no SRAM since it is not present on the target ML605 FPGA board [57]. The display is here used to print the different transformations of the image as well as the final result as illustrated by Figure 6.9. The processor template is also similar however there can be several instances of the following associated to the processors: shared memory, local memory and co-processor. The *DSE-bound* settings given as parameters for the exploration were:

- There can be between one to four MicroBlazes;
- There are two different memory types available: DDR and BRAM.
- The *Face detection* task can also be duplicated to exploit data-parallelism;
- The target is a Xilinx ML605 FPGA board [57].
- The N_X variables that set the scalability of the DSE were all set to -1, meaning that no pruning was performed.

Again, no expert level constraint is specified in this case.

Data mapping exploration

In order to show the influence of the data mapping on the performance, an exploration was performed where the task mappings remained constant. In this exploration, no exploration for hardware acceleration is performed and data mappings were specified manually, so that they remain constant between the different tested architectures. Also the hypothesis was made that all the data can be mapped onto one single memory, thus no constraint was put on the size of the memories.

The results are shown in Table 6.9 and in Figure 6.12. The task mapping notation is similar to the one explained for the MJPEG. Tasks are in the order of the numbers given in Figure 6.8. The memory mapping is specified as follows: there are as many numbers as there are communication channels in application — the order is the same that in Table 6.7 — and the number specifies the memory, in the same order as specified in the previous column, i.e. in this case 0 means DDR and 1 means BRAM. The speedup of each solution is computed from the worst solution, in this case the monoprocessor solution where all the data are mapped in DDR.

The results clearly show the influence of the data mapping on the global performance of the application. For instance in the monoprocessor architecture, the difference in performance between the full-BRAM mapping and the full-DDR mapping is of a factor 3. Hence the importance of having a good data-mapping algorithm. We can also observe that the links corresponding to the classifier — the numbers from the second position up to the seventh — have the most influence. This due to the fact that they are accessed many times — especially the first stages — and thus it represents a huge amount of data. According to one of our profilings, the data from the first stage are

accessed 39327 times for a complete detection of one image. This represents over 130 MBytes of data for the first stage alone. Consequently, when the classifiers are mapped on BRAM, the application is much faster.

We can also notice in the results that the number of processors has little influence in the results. As said before, in our experiments there is only one iteration of the application — i.e. we are only processing one picture —, consequently no task-parallelism can be exploited with the exception of the *Classifier loading* task which can be performed in parallel with the image preprocessing. This is why we have a gain in performance when adding a second processor, but then the execution times remain more or less the same when adding a third or a fourth processor. This shows the need to explore heterogeneous architectures and task-mapping in the domain of embedded systems when no parallelism is available.

Table 6.9: Results of data mapping exploration for the face detection application. It shows the impact of data mapping on the final performances.

Architecture	**Task Mapping**	**Sesame Estimation**	**Memories**	**Memory Mapping**	**Speedup**
MB	0 0 0 0 0 0 0 0 0 0 0 0	3572076416	DDR BRAM	1 1 0 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0	1,69
MB	0 0 0 0 0 0 0 0 0 0 0 0	4037332416	DDR BRAM	1 0 1 0 1 1 1 1 1 1 1 1 0 0 0 0 0 0	1,49
MB	0 0 0 0 0 0 0 0 0 0 0 0	4525249535	DDR BRAM	1 0 0 0 1 1 1 1 1 1 1 1 0 0 0 0 0 0	1,33
MB	0 0 0 0 0 0 0 0 0 0 0 0	2116825535	DDR BRAM	0 1 1 1 1 1 1 0 0 0 0 0 1 1 1 0 0 0	2,85
MB	0 0 0 0 0 0 0 0 0 0 0 0	3952468416	DDR BRAM	0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0	1,53
MB	0 0 0 0 0 0 0 0 0 0 0 0	6025465535	DDR BRAM	0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 0 0	1
MB	0 0 0 0 0 0 0 0 0 0 0 0	2058617535	DDR BRAM	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1	2,93
MB	0 0 0 0 0 0 0 0 0 0 0 0	6031585535	DDR BRAM	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	1
MB MB	0 1 1 0 1 0 0 0 1 0 0	3326095018	DDR BRAM	1 1 0 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0	1,81
MB MB	0 1 1 0 1 0 0 0 1 0 0	3704023550	DDR BRAM	1 0 1 0 1 1 1 1 1 1 1 1 0 0 0 0 0 0	1,63
MB MB	0 1 1 0 1 0 0 0 1 0 0	4142594514	DDR BRAM	1 0 0 0 1 1 1 1 1 1 1 1 0 0 0 0 0 0	1,46
MB MB	0 1 1 0 1 0 0 0 1 0 0	1781351353	DDR BRAM	0 1 1 1 1 1 1 0 0 0 0 0 1 1 1 0 0 0	3,39
MB MB	0 1 1 0 1 0 0 0 1 0 0	3698600727	DDR BRAM	0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0	1,63
MB MB	0 1 1 0 1 0 0 0 1 0 0	5618087075	DDR BRAM	0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 0 0	1,07
MB MB	0 1 1 0 1 0 0 0 1 0 0	1806071993	DDR BRAM	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1	3,34
MB MB	0 1 1 0 1 0 0 0 1 0 0	5624207075	DDR BRAM	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	1,07
MB MB MB	0 1 1 2 2 2 1 2 2 0 2	2084067280	DDR BRAM	1 1 0 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0	2,89
MB MB MB	0 1 1 2 2 2 1 2 2 0 2	3670913536	DDR BRAM	1 0 1 0 1 1 1 1 1 1 1 1 0 0 0 0 0 0	1,64
MB MB MB	0 1 1 2 2 2 1 2 2 0 2	4109484500	DDR BRAM	1 0 0 0 1 1 1 1 1 1 1 1 0 0 0 0 0 0	1,47
MB MB MB	0 1 1 2 2 2 1 2 2 0 2	1658972496	DDR BRAM	0 1 1 1 1 1 1 0 0 0 0 0 1 1 1 0 0 0	3,64
MB MB MB	0 1 1 2 2 2 1 2 2 0 2	3698584613	DDR BRAM	0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0	1,63
MB MB MB	0 1 1 2 2 2 1 2 2 0 2	5619524107	DDR BRAM	0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 0 0	1,07
MB MB MB	0 1 1 2 2 2 1 2 2 0 2	1675077145	DDR BRAM	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1	3,6
MB MB MB	0 1 1 2 2 2 1 2 2 0 2	5625644107	DDR BRAM	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	1,07
MB MB MB MB	0 2 3 1 0 2 1 2 3 1 1	3225915241	DDR BRAM	1 1 0 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0	1,87
MB MB MB MB	0 2 3 1 0 2 1 2 3 1 1	3683237804	DDR BRAM	1 0 1 0 1 1 1 1 1 1 1 1 0 0 0 0 0 0	1,64
MB MB MB MB	0 2 3 1 0 2 1 2 3 1 1	4121808768	DDR BRAM	1 0 0 0 1 1 1 1 1 1 1 1 0 0 0 0 0 0	1,46
MB MB MB MB	0 2 3 1 0 2 1 2 3 1 1	1647617440	DDR BRAM	0 1 1 1 1 1 1 0 0 0 0 0 1 1 1 0 0 0	3,66
MB MB MB MB	0 2 3 1 0 2 1 2 3 1 1	3666889069	DDR BRAM	0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0	1,64
MB MB MB MB	0 2 3 1 0 2 1 2 3 1 1	5547342464	DDR BRAM	0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 0 0	1,09
MB MB MB MB	0 2 3 1 0 2 1 2 3 1 1	1607916444	DDR BRAM	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1	3,75
MB MB MB MB	0 2 3 1 0 2 1 2 3 1 1	5562252966	DDR BRAM	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	1,08

Automated Data Mapping Exploration & HW Accelerators.

In order to evaluate the efficiency of our algorithm, we have launched another exploration, this time with fully-automatized memory mapping decisions, and with the possibility for hardware acceleration. Here the tasks specified by the designer as accelerable through hardware are *Contrast enhancement* and *Face detection*, noted HEQ and DET in Table 6.10 respectively. The accelerators were already present in the database and thus no exploration was performed. Also the accelerator for the *Face detection* task does not accelerate the whole task but simply a part of the classifier computation. In this experiment, we performed several iterations of the program by executing it on several images, in a fashion similar to frames coming from the video stream of a camera. A consequence of having multiple iterations is that it reduce the importance of the loading of the classifiers (the *Classifier loading* task), since this operation has only to be performed once for all the images.

Table 6.10 and Figure 6.13 show the best results based on performances for a given hardware architecture. The speedup is computed from the worst case found in this given selected results, which is the full software monoprocessor solution. We can see that it provides results with good performances and that our data-mapping algorithm makes decisions that minimize the communication cost.

During this exploration, we generated 122 architectures in about 4 seconds which were then evaluated with Sesame in 230 seconds, resulting in a total time of exploration of 234 seconds. For an application that size, exhaustive mapping exploration is no longer possible: for two processors the task mapping would give 2048 solutions, with two memories and 18 communication channels there would be 262144 data mapping possibilities resulting in a total of over 536 millions solutions to evaluate, if the exploration for both data and task mappings were exhaustive.

Parallelism Exploration

Hypotheses Another series of results were launched in order to study the influence of the exploitation of the parallelism. For that we explored two different kinds of parallelisms: data-parallelism and task-parallelism. For the data-parallelism this was done by duplicating the task *Face detection* three times, since it is the most resource-consuming task. To explore the pipelining, we perform several iterations of the program, in a similar fashion of the previous experiment. The accelerators are the same than in the previous exploration. In these results, some architectures have a processor that has two coprocessors, one for the *Face detection* task and one for the *Contrast enhancement*. These processors are noted *HEQDET* in the results. In this exploration, we took as hypothesis that all communications would fit on BRAM memories which have a latency of one cycle.

Results A selection of results of the exploration are shown in Table 6.11 and in Figure 6.14. We can see that this time the number of processors have an impact over the final performances, as the architectures with more processors provides better performances.

When there is more than one processor, we can observe that duplication of the *Face detection* task provide better performances. however we can see in some cases that the version with duplicated tasks is slower than the version with no duplicated tasks: this happens when the architecture contains at least one processor that do not have a coprocessor for *Face detection* task. This is because our tool favors the mapping of duplicated tasks on different processors instead of the most efficient ones, otherwise the benefits of duplication would disappear. These results were kept only for this exploration experiment but in a real-case, our tool would check that there are enough coprocessors given the number of times a task was duplicated, and would discard results that do not fit this hypothesis. If we left out these cases, the speedups obtained over similar architectures with no duplication is between 1.42 and 2.85, which is satisfying since these speedups were obtained with no additional resources costs.

The best architecture obtained yields a speedup of 10.15 over the worst case, the full software monoprocessor solution. This best solution corresponds to the architecture with the maximum number of processors and where there is an accelerator for the *Contrast enhancement* task and one accelerator for each of the duplicated *Face detection* tasks. Given a frequency of 100 MHz for the target architecture, that would correspond to the processing of 0.59 images per second.

Those results were obtained in four runs of the tools, one for each different configurations. In total there were 112 architectures generated — 4×28 — in 742 seconds. The generation times were similar for each of the four runs, only the simulation times differ as the implementations with iterations were taking the most time, since their execution traces were bigger and consequently took longer to simulate.

Conclusion

This case study showed the efficiency of our data-mapping algorithm. It also shows that the exploitation of data parallelism is useful and can provide good performances without increasing the cost.

6.4 Conclusion

With these results, we have shown that our framework is capable of providing efficient solutions very quickly. With the two studied applications, we have validated all the exploration methods we used: task mapping with the Hungarian algorithm, hardware exploration through HLS-based method, data-parallelism exploration through task duplication and successfully taken into account the data mapping. We have also compared our performance with the Daedalus framework, which accepts the same application types as us — i.e. SANLP — and shown that we were much faster in spite of the loss in optimality of the results. Finally we have also shown that our performance estimation are close to the ones of on-target implementation, thus proving the accuracy of our estimators.

Besides the efficiency of the solutions it provides, our framework also gives a set of methods and tools — even though some of them still need to be finalized — that

could simplify greatly the design by providing solutions that are far from obvious in a domain as complex as the H-MPSoC one.

Table 6.10: Results of the automated mapping exploration.

Architecture	Mapping	Sesame Est.	Memories	Memory Mapping	Speedup	Slices	BRAM	DSP
MB	0 0 0 0 0 0 0 0 0 0 0 0	23892860176	DDR BRAM	1 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1	1	2103	0	0
MB MB	1 1 1 0 0 1 0 1 1 0 1	19957798877	DDR BRAM	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 1 1 1	1,2	4206	0	0
MB MB MB	0 0 2 2 2 0 1 0 1 1 0	14503782376	DDR BRAM	1 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1	1,65	6309	0	0
MB MB MB MB	1 3 2 2 3 2 0 1 0 0 3	14129193167	DDR BRAM	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 1 1 1	1,69	8412	0	0
HEQ	0 0 0 0 0 0 0 0 0 0 0 0	17770105070	DDR BRAM	1 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1	1,34	9308	0	256
MB HEQ	1 1 1 0 1 1 0 0 1 0 1	14684103028	DDR BRAM	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 1 1 1	1,63	11411	0	256
MB MB HEQ	1 1 2 0 2 2 2 1 2 0 1	13708065693	DDR BRAM	1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1	1,74	13514	0	256
MB MB MB HEQ	0 2 1 1 3 3 2 0 3 2 2	14955415396	DDR BRAM	1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1	1,6	15617	0	256
DET	0 0 0 0 0 0 0 0 0 0 0 0	13197892720	DDR BRAM	1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1	1,81	2103	0	0
MB DET	0 0 0 1 0 0 1 1 0 1 0	10711195459	DDR BRAM	1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1	2,23	4206	0	0
MB MB DET	0 0 1 1 0 1 2 1 2 2 0	10191504361	DDR BRAM	1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1	2,34	6309	0	0
MB MB MB DET	1 3 0 0 2 0 2 1 2 3 3	8661824052	DDR BRAM	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1	2,76	8412	0	0
DET HEQ	1 1 1 0 1 1 0 0 1 0 1	9482325330	DDR BRAM	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 1 1 1	2,52	11411	0	256
MB DET HEQ	2 2 0 1 2 0 0 0 0 1 2	8916949321	DDR BRAM	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1	2,68	13514	0	256
MB MB DET HEQ	3 0 1 1 3 0 2 0 2 2 0	12554660131	DDR BRAM	1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1	1,9	15617	0	256

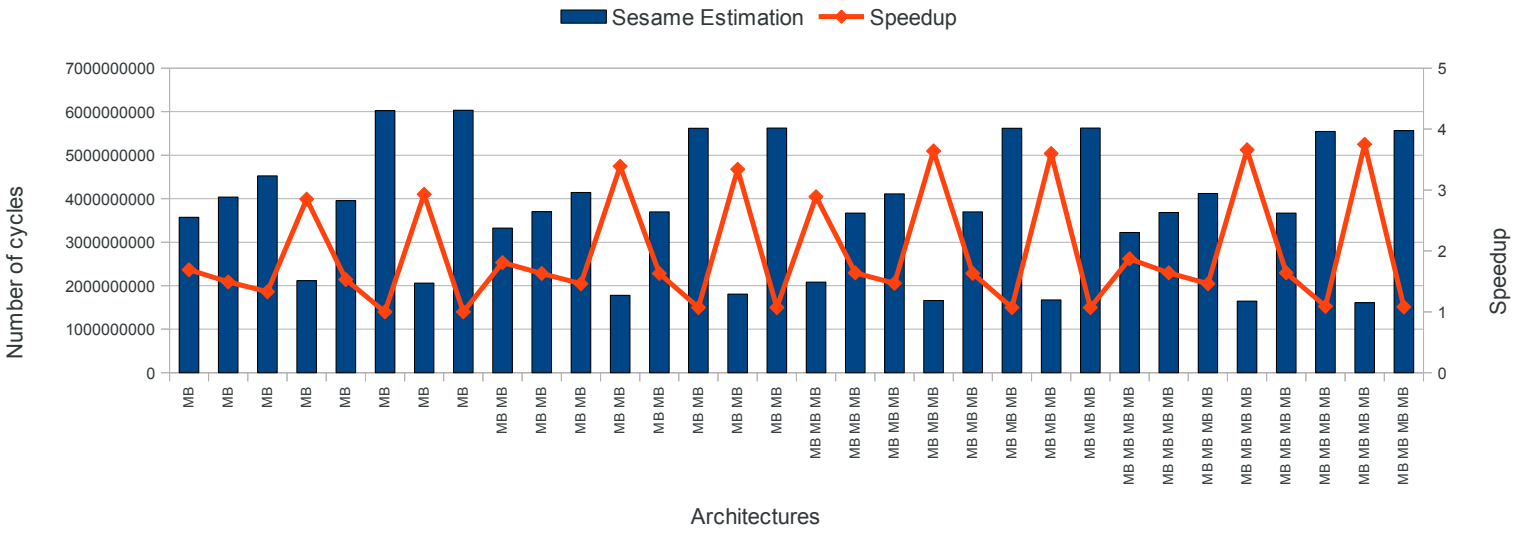

Figure 6.12: Results of data mapping exploration.

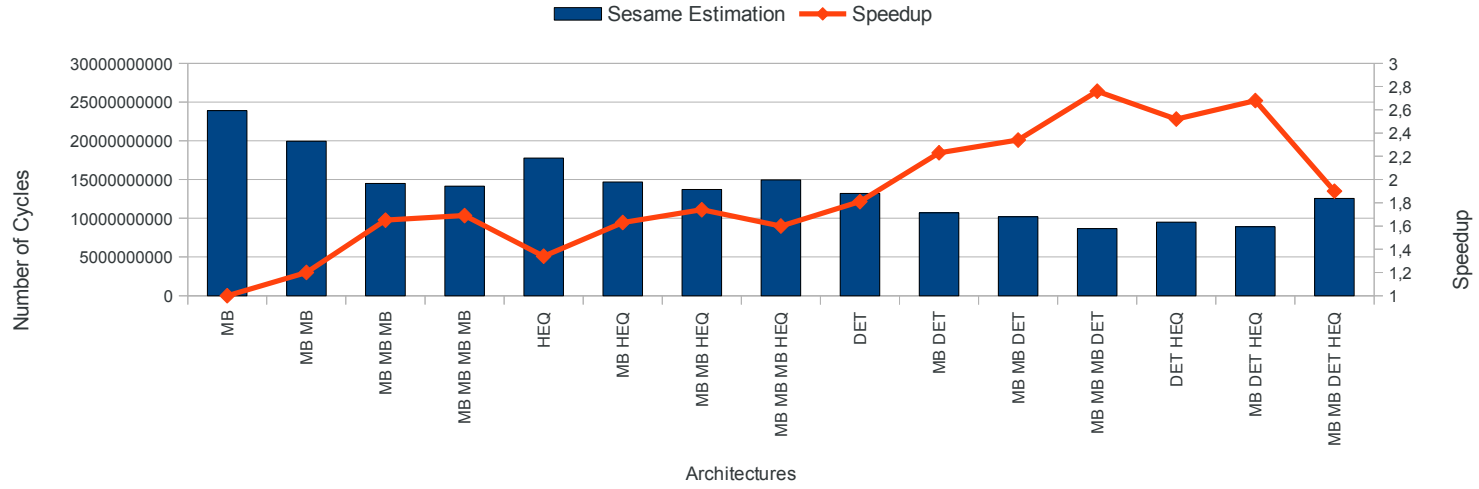

Figure 6.13: Results of the automated exploration.

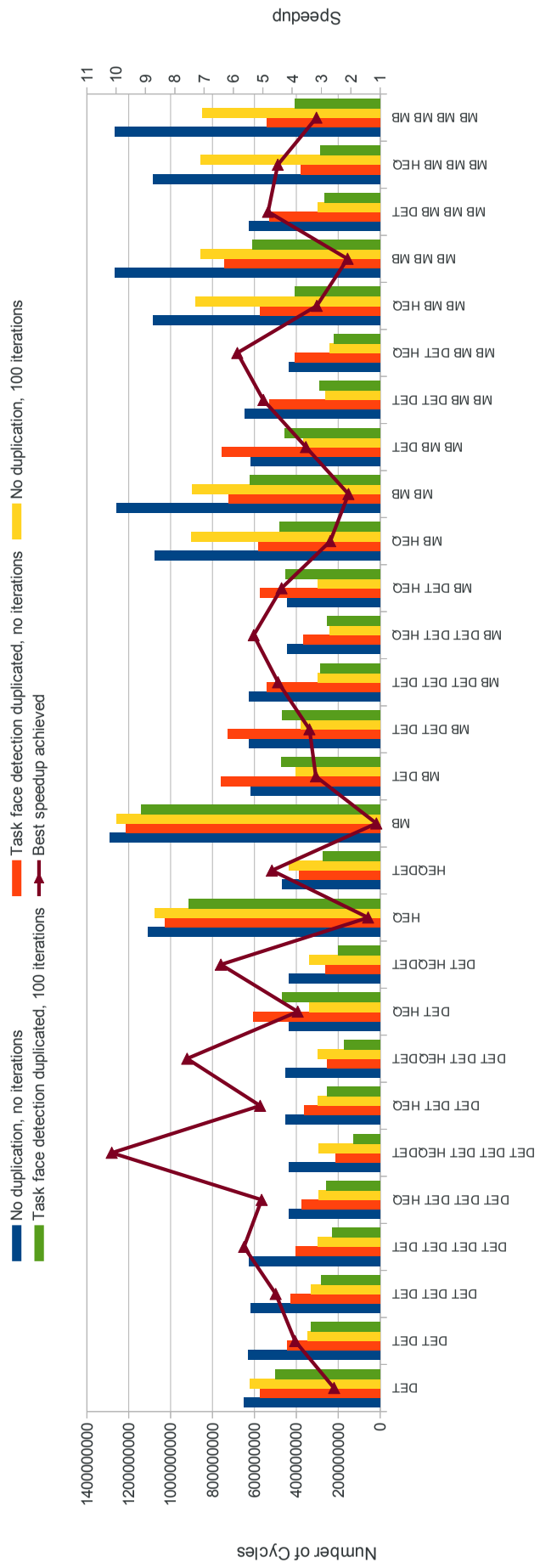

Figure 6.14: Results of parallelism exploration.

Table 6.11: Results of the parallelism exploration. For each implementation the speedpus are given in comparison with the implementation where no duplication and no iteration are performed. The last column gives the best speedup obtained in comparison with the worst case i.e. the software monoprocessor implementation

Architecture	**No duplication, no iterations**	**Task face detection duplicated, no iterations**	**Speedup**	**No duplication, 100 iterations**	**Speedup**	**Task face detection, 100 iterations**	**Speedup**	**Speedup over Software Monoprocessor**
DET	648692376	571498564	1,14	619924068	1,05	500610435	1,3	2,57
DET DET	630080341	445049486	1,42	347841120	1,81	330792161	1,9	3,9
DET DET DET	617066053	426843467	1,45	332039773	1,86	282534817	2,18	4,56
DET DET DET DET	627894602	402234242	1,56	296154442	2,12	228586432	2,75	5,64
DET DET DET HEQ	434287452	375510221	1,16	293102088	1,48	255455410	1,7	5,04
DET DET DET HEQDET	434287466	211484481	2,05	293102088	1,48	126950800	3,42	10,15
DET DET HEQ	451978349	362717459	1,25	295787442	1,53	253308240	1,78	5,09
DET DET HEQDET	451978191	254479220	1,78	295787447	1,53	169990807	2,66	7,58
DET HEQ	434287474	607391886	0,72	337997768	1,28	469893461	0,92	3,81
DET HEQDET	434287374	262266344	1,66	337997768	1,28	200439287	2,17	6,43
HEQ	1105913611	1028719799	1,08	1077145303	1,03	914881997	1,21	1,41
HEQDET	465913595	388719783	1,2	437145287	1,07	274881981	1,69	4,69
MB	1288692392	1211498580	1,06	1259924084	1,02	1140610451	1,13	1,13
MB DET	617066001	761165310	0,81	404311812	1,53	470692108	1,31	3,19
MB DET DET	624472333	726862859	0,86	378465102	1,65	468232920	1,33	3,41
MB DET DET DET	627894620	541386624	1,16	296154442	2,12	287515040	2,18	4,48
MB DET DET HEQ	445188017	367223546	1,21	242491246	1,84	252955577	1,76	5,31
MB DET HEQ	442422871	574138512	0,77	295787447	1,5	451815377	0,98	4,36
MB HEQ	1074359536	580446531	1,85	899322368	1,19	477489550	2,25	2,7
MB MB	1257138247	723549718	1,74	897642973	1,4	620307541	2,03	2,08
MB MB DET	617138139	755276464	0,82	364606042	1,69	456425125	1,35	3,53
MB MB DET DET	645061104	528162395	1,22	258998202	2,49	288775126	2,23	4,98
MB MB DET HEQ	434287416	405103199	1,07	242491246	1,79	219445789	1,98	5,87
MB MB HEQ	1085117147	574138432	1,89	881987256	1,23	407709321	2,66	3,16
MB MB MB	1264472209	742959675	1,7	858087399	1,47	610185510	2,07	2,11
MB MB MB DET	627895808	528521909	1,19	296154455	2,12	267211289	2,35	4,82
MB MB MB HEQ	1085188035	380156441	2,85	855817133	1,27	286710851	3,78	4,49
MB MB MB MB	1267894636	540089421	2,35	849519052	1,49	407087132	3,11	3,17

7 Conclusion

7.1 Summary

In this thesis, we have presented a framework to make possible the design of Heterogeneous Multiprocessor Systems-on-Chip on FPGA, which are far from current practices in embedded system industry. Indeed designing such systems is still a challenge due to the complexity of H-MPSoC architecture which combines parallelism and heterogeneous programming, paradigms that remain major obstacles to efficient designs. Few designers have both these expertises and consequently a new generation of tools is needed at an even higher-level of specifications in order to abstract the lower-level design complexity. In addition to the programming difficulties, the growing complexity of those systems is also a problem for their design as the number of design options explodes and can no longer be evaluated manually. Moreover the optimization of such architectures lies on design choices, which are usually not straightforward. As a matter of fact, the efficiency of the final design is strongly dependent on memory mapping, number of processors, interconnect scheme and hardware accelerator choices.

In this work, we have proposed a solution to these problems, by providing designers with a design flow that allows them to choose their involvement in the design: they can provide a few basic constraints and let the tool explore the solutions or can provide extra-constraints, resulting from their skills and their knowledge of the system under design, letting the tool deals only with the most tedious and time-consuming tasks. We have presented in this thesis, several contributions to achieve this goal.

First, the use of a template based approach provides a simple way to specify design constraints. It can specify a generic architecture description to cope with the lack of underlying processor architecture model, contrary to standard in fixed architectures — e.g. ARM-based — for embedded systems. This architecture contains domain specific constraints and is used as a basis for the DSE. The template's multiple levels of specifications allow designers to express the design constraints according to their levels of expertise and on how much they want to get involve in the design. Based on this level of constraints, the tool will adapt its design flow, whether dealing with only the tedious steps or going further by taking most of the design decisions.

An automated and scalable design space exploration method, that relies on fast and accurate estimators for cost and performance. This method allows to quickly evaluate a great number of design options and accordingly take the best decisions. It brings an

answer to the increasing size of the design space due to the complexity H-MPSoC. The DSE flow explores the following dimensions: the number and types of processors, buses, memories, task mappings, data mappings, data-parallelism and hardware accelerators.

In order to perform the latter, our work also introduces a methodology for exploring tradeoff between cost and performances in hardware accelerators through HLS. Based on the IP characteristics and the use of analytical models, it can provide accurate estimations of the logic resource cost thus avoiding a time-consuming logic synthesis. This time-saving technique allows the integration of hardware accelerators exploration inside the main DSE loop. We also provide a way to speed up the system with a minimal cost, by exploiting data parallelism through task duplication.

Finally, the use of MDE approaches enables model transformation, code generation and model checking in the design. It brings a solution to the problem of lack of standard API for FPGA designs by abstracting the design specifications from specific hardware implementation details. As a consequence, designs becomes more easily portable from one target to another. We also implemented databases populated from previous designs and that contain several design elements that can be reused in future designs. This can greatly speed-up the design phase as it provides already-tested and consequently bug-free solutions.

In this thesis, we have proposed a tool flow that simplifies the design through the automation of cumbersome design tasks, enables reusability and portability through MDE approaches and takes into account the designer expertise to provide better solutions. This approach has been successfully validated with real case applications. Our approach is a first step towards an environment for embedded system design based on FPGA. This kind of tool is particularly required by SME that expect development time and complexity close to existing solutions for processor-based platform. In this work, we have demonstrated that the proposed framework enables the exploration of hundreds of configurations in a few seconds. Short exploration time is important but more than anything it means that incremental design methods are possible. So it opens new perspectives since the choice of the architecture template including memory organization, different task splitting or task decomposition can be tested by designers who can have fast feedbacks to evaluate their choices.

7.2 Perspectives

If we take the description of the ideal framework flow described in Section 2.1, we observe that some steps still need to be considered. So to extend further our work, we propose the followings:

- Take into account power consumption during DSE. This include the development of a model for power consumption that can evaluate the energy cost of components. These estimations can be based on the power consumption models of the Open-PEOPLE project [46] which are also based on AADL thus facilitating their integrations into our framework. Another way to enhance power saving, would be

to take into account dynamic reconfigurabition, thus allowing the use of smaller, less power-consuming FPGA.

- Extend the scalability of the designed architectures in order to enable the support of manycore architectures. Such support requires, among other things, to include the possibility to use a NoC. Our template-based approach can be adapted to the support of manycore by using it to define the basic repeatable pattern used to set the scalability.

- Implement the graphical user interface for configuring the inputs of the framework described in Section 5.5.1. This GUI could also be used to display the results of the exploration and allows designers to select the architecture they want.

- Populate the databases by providing more templates and components to support other architectures. This also includes the specification of more analytical models for HLS-based cost estimation during hardware accelerators exploration in order to extend the supported targets.

- To adapt the flow of our framework to other MoCs. This would allows to broaden the range of applications that can be accepted as input in our tool. However that would also require a substantial amount of additional development, as the KPN MoC strongly constrains the application and consequently resolve a consequent number of problems, especially for communication and synchronization.

- Exploit the proposed framework as an opportunity for a technology transfer. That would requires additional software development to go from the proof of concept framework we proposed to an industrial Computer-Aided Design (CAD) tool.

8 Bibliography

[1] TI OMAP Processors presentation webpage. `http://www.ti.com/general/docs/gencontent.tsp?contentId=46946`. Last accessed: 03/12/2012. 5

[2] Xilinx Zynq Platform documentation webpage. `http://www.xilinx.com/support/documentation/zynq-7000.htm`. Last accessed: 03/12/2012. 6

[3] K. Benkrid, A. Akoglu, C. Ling, Y. Song, Y. Liu, and X. Tian. High Performance Biological Pairwise Sequence Alignment: FPGA versus GPU versus Cell BE versus GPP. *International Journal of Reconfigurable Computing*, 2012, 2012. 6, 30, 72

[4] M. Thompson, H. Nikolov, T. Stefanov, A.D. Pimentel, C. Erbas, S. Polstra, and E.F. Deprettere. A framework for rapid system-level exploration, synthesis, and programming of multimedia mp-socs. In *Proceedings of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, pages 9–14. ACM, 2007. 11, 12, 39, 89

[5] S. Verdoolaege, H. Nikolov, and T. Stefanov. PN: A tool for improved derivation of process networks. *EURASIP Journal on Embedded Systems*, 2007(1):19–19, 2007. 12, 40

[6] H. Nikolov, T. Stefanov, and E. Deprettere. Multi-processor system design with ESPAM. In *CODES+ ISSS'06*, pages 211–216, 2006. 12, 40

[7] A.D. Pimentel, C. Erbas, and S. Polstra. A systematic approach to exploring embedded system architectures at multiple abstraction levels. *Computers, IEEE Transactions on*, 55(2):99–112, 2006. 12, 41, 48, 68

[8] J. Keinert, T. Schlichter, J. Falk, J. Gladigau, C. Haubelt, J. Teich, M. Meredith, et al. SystemCoDesigner—an automatic ESL synthesis approach by design space exploration and behavioral synthesis for streaming applications. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 14(1):1–23, 2009. 13, 14

[9] J. Falk, C. Haubelt, and J. Teich. Efficient representation and simulation of model-based designs in systemc. In *Proceedings of the International Forum on Specification & Design Languages (FDL'06)*, pages 129–134, 2006. 13

[10] Forte Cynthesizer. `http://www.forteds.com/products/cynthesizer.asp`. 13

[11] S. Shibata, S. Honda, H. Tomiyama, and H. Takada. Advanced SystemBuilder: A tool set for multiprocessor design space exploration. *SoC Design Conference (ISOCC), 2010 International*, pages 79 – 82, 2010. 14, 15

[12] M. Rashid, F. Ferrandi, and K. Bertels. hArtes design flow for heterogeneous platforms. In *Quality of Electronic Design, 2009. ISQED 2009. Quality Electronic Design*, pages 330–338. IEEE, 2009. 15, 16

[13] S. Vassiliadis, S. Wong, G. Gaydadjiev, K. Bertels, G. Kuzmanov, and E.M. Panainte. The MOLEN Polymorphic Processor. *Computers, IEEE Transactions on*, 53(11):1363–1375, 2004. 16

[14] A. Fraboulet, T. Risset, and A. Scherrer. Cycle Accurate Simulation Model Generation for SoC Prototyping. *Computer Systems: Architectures, Modeling, and Simulation*, pages 255–269, 2004. 16

[15] Y. Yankova, G. Kuzmanov, K. Bertels, G. Gaydadjiev, Y. Lu, and S. Vassiliadis. DWARV: Delftworkbench Automated Reconfigurable VHDL Generator. In *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on*, pages 697–701. IEEE, 2007. 16

[16] S. Ha, S. Kim, C. Lee, Y. Yi, S. Kwon, and Y. Joo. Peace: A hardware-software codesign environment for multimedia embedded systems. *ACM Transactions on Design Automation of Electronic Systems*, 12(3), 2007. 17, 18

[17] J.T. Buck, S. Ha, E.A. Lee, and D.G. Messerschmitt. Ptolemy: A framework for simulating and prototyping heterogeneous systems. *International Journal of Computer Simulation*, 4:155–182, 1994. 17, 35

[18] Xilinx Platform Studio website. `http://www.xilinx.com/tools/xps.htm`. Last accessed: 26/11/2012. 18

[19] L. Moss, H. Guérard, G. Dare, and G. Bois. Rapid Design Exploration on an ESL Framework featuring Hardware-Software Codesign for ARM Processor-based FPGA's. *Space*, 1, 2012. 18

[20] P. Coussy, C. Chavet, P. Bomel, D. Heller, E. Senn, and E. Martin. *GAUT: A High-Level Synthesis Tool for DSP applications*. Springer, 2008. 23, 53

[21] Xilinx *Fast Simplex Link* documentation. `http://www.xilinx.com/products/ipcenter/FSL.htm`. 23

[22] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J.H. Anderson, S. Brown, and T. Czajkowski. LegUp: high-level synthesis for FPGA-based processor/accelerator systems. In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, pages 33–36. ACM, 2011. 24

[23] NIOS C2H Compiler's presentation webpage. `http://www.altera.com/products/ip/processors/nios2/tools/c2h/ni2-c2h.html`. 24

[24] CyberWorkBench webpage. `http://www.nec.com/en/global/prod/cwb/`. Last accessed: 05/12/2012. 25

[25] C. Pilato and F. Ferrandi. Bambu: A Free Framework for the High Level Synthesis of Complex Applications. *University Booth of DATE 2012. DATE'12.*, 2012. 25

[26] Berkeley Design Technology. An independent evaluation of the autoesl autopilot high-level synthesis tool., 2010. 26

[27] Marcio F. S. Oliveira, Eduardo W. Brião, Francisco A. Nascimento, and Flávio R. Wagner. Model driven engineering for MPSOC design space exploration. In *Proceedings of the 20th annual conference on Integrated circuits and systems design*, SBCCI '07, pages 81–86. ACM, 2007. 26, 27

[28] R.B. Atitallah, E. Piel, S. Niar, P. Marquet, and J.L. Dekeyser. Multilevel MPSoC Simulation using an MDE Approach. In *SOC Conference, 2007 IEEE International*, pages 197–200. IEEE, 2007. 27

[29] Object Management Group Management Group. A UML Profile for MARTE, Beta 1, 2007. 27

[30] J. Vidal, F. De Lamotte, G. Gogniat, P. Soulard, and J.P. Diguet. A co-design approach for embedded system modeling and code generation with UML and MARTE. In *Design, Automation & Test in Europe Conference & Exhibition, 2009. DATE'09.*, pages 226–231. IEEE, 2009. 28

[31] J. Vidal. *Dynamic and Partial Reconfigurable Embedded Systems Design with UML*. PhD thesis, Université de Bretagne Sud, 2010. 28

[32] Eclipse framework website. `http://www.eclipse.org`. 29, 73

[33] R. Dafali. *Conception des réseaux sur puce reconfigurables dynamiquement*. PhD thesis, Université de Bretagne Sud, 2011. 33

[34] G. Kahn. The semantics of a simple language for parallel programming. *Information processing*, 74:471–475, 1974. 35

[35] S.L. Graham, P.B. Kessler, and M.K. Mckusick. Gprof: A call graph execution profiler. *ACM Sigplan Notices*, 17(6):120–126, 1982. 35

[36] Xilinx, OS and Libraries Document Collection (UG 643). `http://www.xilinx.com/support/documentation/sw_manuals/xilinx12_3/oslib_rm.pdf`. 38

[37] H.L. Muller. *Simulating computer architectures*. PhD thesis, Universiteit van Amsterdam, 1993. 41

[38] J.E. Coffland and A.D. Pimentel. A software framework for efficient system-level performance evaluation of embedded systems. In *Proceedings of the 2003 ACM symposium on Applied computing*, pages 666–671. ACM, 2003. 41

[39] Online OpenCores library. `http://opencores.org/`. 44, 52

[40] Xilinx. Virtex-5 Family Overview (DS100). 2006. 44

[41] H.W. Kuhn. The hungarian method for the assignment problem. *Naval research logistics quarterly*, 2(1-2):83–97, 1955. 51, 66

[42] Van-Trinh Hoang. CAO pour la synthèse d'architectures multiprocesseurs: Interface d'exploration de l'espace de conception, la synthèse d'architecture et une librairie d'IP pré-caractérisées. Master's thesis, Université de Bretagne-Sud, 2011. 52

[43] Xilinx. Virtex-5 FPGA User Guide. *UG190*, 5, 2009. 56

[44] S. Rouxel, G. Gogniat, J-Ph. Diguet, J-L Philippe, and C. Moy. Models Driven Engineering for Distributed Real-Time Embedded Systems. In *From MDD Concepts to Experiments and Illustrations*, pages 111–130 (Chap. 7). Lavoisier, 2006. 72

[45] P.H. Feiler. The architecture analysis & design language (AADL): An introduction. Technical report, DTIC Document, 2006. 72

[46] Open-PEOPLE project descritption. `http://raweb.inria.fr/rapportsactivite/RA2010/dart/uid111.html`. Last accessed: 10/12/2012. 72, 112

[47] R. Ben Atitallah, E. Senn, D. Chillet, M. Lanoe, and D. Blouin. An efficient Framework for Power-Aware Design of Heterogeneous MPSoC. 2011. 72

[48] MicroBlaze Processor Reference Guide. `http://www.xilinx.com/support/documentation/sw_manuals/xilinx13_3/mb_ref_guide.pdf`. 73, 75, 87

[49] Eclipse Modeling Framework Project website. `http://www.eclipse.org/modeling/emf/`. 73

[50] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro. *EMF: Eclipse Modeling Framework*. Addison-Wesley Professional, 2008. 73

[51] Platform Format Specification Reference Manual - Xilinx (UG 642). `http://www.xilinx.com/support/documentation/sw_manuals/xilinx13_2/psf_rm.pdf`, 2011. 81

[52] I. Augé, F. Pétrot, F. Donnet, and P. Gomez. Platform-based design from parallel C specifications. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 24(12):1811–1826, 2005. 85

[53] Xilinx XUPV5-LX110T FPGA Board Documentation. `http://www.xilinx.com/univ/xupv5-lx110t.htm`. Last accessed: 21/08/2012. 86, 88

[54] P. Viola and M. Jones. Rapid object detection using a boosted cascade of simple features. In *Computer Vision and Pattern Recognition, 2001. CVPR 2001. Proceedings of the 2001 IEEE Computer Society Conference on*, volume 1, pages I–511. IEEE, 2001. 94

[55] Face Detection using OpenCV (Open-Source Computer Vision) webpage. `http://opencv.willowgarage.com/wiki/FaceDetection`. Last accessed: 21/11/2012. 94

[56] Y. Freund and R. Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. In *Computational learning theory*, pages 23–37. Springer, 1995. 95

[57] Xilinx ML605 FPGA Board Documentation. `http://www.xilinx.com/products/boards/ml605/reference_designs.htm`. Last accessed: 22/11/2012. 101

List of Publications

International Conferences

1. Y. Corre, J.P. Diguet, D. Heller, and L. Lagadec. A Framework for High-Level Synthesis of Heterogeneous MPSoC. In Proceedings of *the Great Lakes Symposium on VLSI (GLSVLSI 2012)*, Salt Lake City (USA), pages 283–286. ACM, 2012.

2. Y. Corre, V.T. Hoang, J.P. Diguet, D. Heller, and L. Lagadec. HLS-based Fast Design Space Exploration of ad hoc Hardware Accelerators: a Key Tool for MPSoC Synthesis on FPGA. In *International Conference on Design and Architectures for Signal and Image Processing* (DASIP), Karlsruhe (Germany), 2012.

3. Y. Corre, J.P. Diguet, L. Lagadec, D. Heller and D. Blouin. Fast Template-based Heterogeneous MPSoC Synthesis on FPGA. In *Applied Reconfigurable Computing 2013 (ARC 2013)*, Los Angeles (USA), 2013.

Journal

1. L. Lagadec, D. Picard, Y. Corre, and P.Y. Lucas. Experiment Centric Teaching for Reconfigurable Processors. *International Journal of Reconfigurable Computing*, 2011, 2011.

Conferences with no selection

1. Y. Corre, J.P. Diguet, L. Lagadec, and D. Heller. A Framework for Automated Design Space Exploration and Synthesis of Heterogeneous MPSoC, Demonstration at the *University Booth at the Design, Automation & Test in Europe 2012 (UBOOTH DATE 2012)*, Dresden (Germany), 2012.

2. Y. Corre, J.P. Diguet, D. Heller, L. Lagadec, and V.T. Hoang. Complete Heterogeneous MPSoC Synthesis, *GDR SoCSIP*, Lyon (France), 2011.

List of Figures

List of Tables

List of Algorithms

Glossary

AADL Architecture Analysis & Design Language. 70, 78, 79, 81, 108

ALAP As Late As Possible. 64

API Application Programming Interface. 15, 108

ASAP As Soon As Possible. 64

ASIC Application Specific Integrated Circuit. 30

ATL ATL Transformation Language. 71, 79

CABA Cycle-Accurate Byte-Accurate. 15, 18, 19, 21, 28

CAD Computer-Aided Design. 109

DSE Design Space Exploration. 7, 19, 24, 28, 74, 76, 78

DSL Domain Specific Language. 69, 81

DSP Digital Signal Processor. 16

ESL Electronic System-Level. 11

FIFO First-in, First-Out. 33, 35

FPGA Field-Programmable Gate Array. 6, 29, 70

FSL Fast Simplex Link. 73, 86

GPP General Purpose Processor. 6, 16, 30

GPU Grapical Processing Unit. 6

GUI Graphical User Interface. 9

H-MPSoC Heterogeneous Multiprocessor System-on-Chip. 5, 7, 8, 19, 79, 108

HLS High-Level Synthesis. 7, 26, 39, 108

IDCT Inverse Discrete Cosine Transform. 83, 84

IP Intellectual Property. 16, 18, 87

ISS Instruction Set Simulator. 11, 17

KPN Kahn Process Network. 35, 84

LMB Local Memory Bus. 73, 86

LUT Look-Up Tables. 76

MDE Model-Driven Engineering. 8, 28, 69, 70, 81, 108

MoC Model of Computation. 17, 18, 34, 35

MPSoC Multiprocessor System-on-Chip. 5, 28

NoC Network-on-Chip. 33, 73, 109

PLB Processor Local Bus. 73, 85

QVT Query/View/Transformation. 71

SANLP Static Affine Nested-Loop Program. 35, 84, 102

SME Small and Medium Enterprises. 5, 108

SoC System-on-Chip. 5

UML Unified Modeling Language. 69

VLIW Very Long Instruction Word. 34