

TBES: Template-Based Exploration and Synthesis of Heterogeneous Multiprocessor Architectures on FPGA

Youenn Corre, Jean-Philippe Diguet, Dominique Heller, Dominique Blouin, Loïc Lagadec

▶ **To cite this version:**

Youenn Corre, Jean-Philippe Diguet, Dominique Heller, Dominique Blouin, Loïc Lagadec. TBES: Template-Based Exploration and Synthesis of Heterogeneous Multiprocessor Architectures on FPGA. ACM Transactions on Embedded Computing Systems (TECS), 2016, 15 (1), pp.9. hal-01172103

HAL Id: hal-01172103
https://hal.science/hal-01172103

Submitted on 11 Mar 2020

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

TBES: Template-Based Exploration and Synthesis of Heterogeneous Multiprocessor Architectures on FPGA

Final submission to TECS, published as:

Y. Corre, J-Ph. Diguet, D. Heller, D. Blouin, L. Lagadec, *TBES: Template-Based Exploration and Synthesis of Heterogeneous Multiprocessor Architectures on FPGA*, ACM Trans. on Embedded Computing Systems (TECS), Vol. 15 Issue 1, Article. No. 9, Jan. 2016.

Abstract

This paper describes TBES, a software end-to-end environment for synthesizing multi-task applications on FPGAs. The implementation follows a template-based approach for creating heterogeneous multiprocessor architectures. Heterogeneity stems from the use of general-purpose processors along with custom accelerators. Experimental results demonstrate substantial speedup for several classes of applications. Furthermore, this work allows to reduce development costs and save development time, both for the software architect, the domain expert, and the optimization expert. This work provides a framework to bring together various existing tools and optimisation algorithms. The advantages are manifold: modularity and flexibility, easy customization for best fit algorithm selection, durability and evolution over time, and legacy preservation including domain expert's know-how. In addition to the use of architecture templates for the overall system, a second contribution lies upon using high-level synthesis for promoting exploration of hardware IPs. The domain expert, who best knows which tasks are good candidates for hardware implementation, selects parts of the initial application, to be potentially synthesized as dedicated accelerators. As a consequence, HLS general problem turns into a constrained and more tractable issue, and automation capabilities eliminate the need for tedious and error prone manual processes during domain space exploration. The automation only takes place once the application has been broken down into concurrent tasks by the designer, who can then drive the synthesis process with a set of parameters provided by TBES to balance tradeoffs between optimization efforts and quality of results. The approach is demonstrated step by step up to FPGA implementations and executions with an MJPEG benchmark and a complex Viola-Jones face detection application. We show that TBES allows to achieve results with up to 10X speedup, to reduce development times and to widen design space exploration.

Keywords: *Electronic System Level, High-Level Synthesis, Multiprocessor, System-on-Chip.*

1 Introduction

In the domain of embedded systems like in high performance computing, Field-Programmable Gate Arrays (FPGA) and hybrid devices provide configurable resources for hardware (HW) specialization as well as the opportunity to take advantage of available parallelism. Both of these aspects improve the energy efficiency compared to applications running on Graphics Processing Units (GPU) and General-Purpose Processors (GPP). Application domains such as smart cameras [1], advanced acoustics [2] and cognitive radio [3] are typical examples that benefit from FPGAs including reconfiguration capabilities. Moreover embedded systems may soon take further advantage of new FPGA architectures, based on non-volatile memories with power gating capabilities [4], that will bring impressive gains in terms of power consumption.

However, embedded systems also have stringent productivity and small-market constraints. This means that designers might prefer standard architectures and easy to use design tools. Their designs also significantly rely on intensive code and IP reuse strategies. Such practices do not fit with the current complexity of programming heterogeneous hardware/software architectures on FPGAs. This programming and debugging complexity hinders their widespread adoption, despite recent progress in hardware/software interfacing in the domain of embedded systems (e.g. ARM/FPGA Accelerator Coherency Port on the Xilinx Zynq platform [5]) and HPC (IBM/Altera Coherent Accelerator Processor Interface [6]). The main reasons for these shortcomings are a lack of underlying processor architectures and a limited design reuse.

To cope with these issues, we rely on templates for Heterogeneous Multiprocessor SoC (HMPSoC) that introduce a conceptual layer, which makes the technology accessible for designers and CAD tools. The design space is constrained by these templates, and captured within flexible architecture models (specific to a given application domain), following a well-defined model of computation. This approach is based on the observation that exhaustive exploration of architecture models is useless since in practice, predefined models are well identified for given application domains. For instance, the authors of [7] propose architecture skeletons for image

processing, which can be mapped to different FPGA family targets, and updated when new targets become available. Such an approach allows to improve resource and performance estimations, and the use of fast High-Level Synthesis (HLS) tools within the Design Space Exploration (DSE) loop. Memory mapping is another tedious task for embedded system designers, and template-based methods enable the development of efficient heuristics to automate this task. Finally, code and scripts generation is simplified and allows the elimination of time-consuming low level tasks. In addition to system synthesis, the formalization and constraint of the design space through models provide a better control of the exploration of design properties such as the number and type of processors, the number, type and size of memories, the tasks and data mappings, and the coprocessors choices.

FPGAs offer more power-efficient solutions than GPPs, and support spatial parallelism leading to higher performance. However, programming for such devices remains less tractable than writing code for software functions. As a consequence, it makes sense to keep portions of an application as code running on a GPP. The main benefits are: an improved adaptability, a wide reuse of legacy code, and a good readability. On the other hand, when considering race conditions, FPGAs implementation is needed to achieve spatial execution (trading extra hardware resources for execution time). This observation advocates the use of heterogeneous systems [8], which best combine the sought characteristics of both solutions: flexibility and performances.

Fig. 1 illustrates the positioning of heterogeneous MPSoC designs in terms of flexibility, energy efficiency and design cost for various implementation platforms (GPP, Application-Specific Integrated Circuit (ASIC), MPSoC on FPGA, etc.). Systems implemented on GPPs are very flexible but suffer from low energy efficiency. On the opposite, ASICs are quite efficient but expensive to design. MPSoCs on FPGAs are a good compromise, but they remain complex to develop. Hence, the objective of the Template-Based Synthesis (TBES) approach and tool, presented in this paper, is to reduce the effort inherent to the development of FPGA-based systems, in order to benefit from efficiency gains provided by heterogeneity (i.e. coprocessors). We first start by presenting related work in Section 2 to compare our solution with the state of the art and to introduce our contributions. Section 3 presents the definition of the template-based architecture approach, and Section 4 describes the different steps of our design flow and TBES framework. In Section 5, we show how TBES is efficiently applied: first, on a classical Motion JPEG (MJPEG) application and second, on a complex Viola-Jones application for face detection. Finally, we conclude and present perspectives.

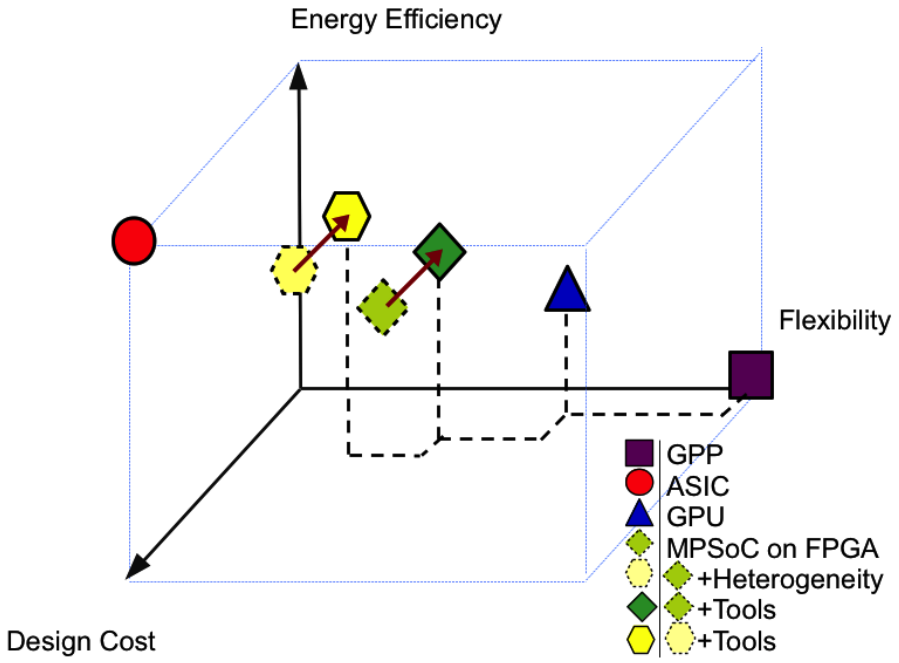


Figure 1: Positioning of heterogeneous MPSoC designs in terms of flexibility, energy efficiency and design cost for various execution platforms.

2 Related work and contributions

2.1 Related work

Over the years, numerous Electronic System-Level (ESL) design tools have been proposed for HMPSoC design, and a selection of these tools is presented in Table 1. We take as reference a hypothetical ideal ESL tool, regardless of its feasibility, and compare it with existing tools including TBES in order to state our contributions.

Frameworks	Inputs			Design Space Exploration					Outputs		
	Application Specifications	Hardware	Constraints	Automated Exploration		Scalable	Explored dimensions	Estimations	Code generation	Hardware	Software
				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	Arch. Model, Task & Data Mapping, Task & Data Parallelism, HW/SW partitioning, Task granularity and scheduling. Metrics: Performance, area, power	Fast and accurate	Yes	Synthesizable architecture	Adapted SW code
TBES (our tool)	C/C++ code in SANLP	Architecture template in AADL	Provided as part of the architecture template	Yes	Yes (with HLS)	Yes (pruning + constraints)	Task & Data Mapping, Task Parallelism, HW/SW partitioning. Metrics: Performance, area	Trace-based Simulation; Model-based cost estimation	Yes	Synthesizable arch. + backend tools files	Adapted code
Daedalus [9]	C/C++ code in SANLP	Component definitions in Pearl description language / XML for the architecture	XML files / GUI	Yes	No	Yes	Task Mapping, HW/SW partitioning, Scheduling. Metrics: Performance, area	Trace-based simulation	Yes	Synthesizable arch. + backend tools files	Adapted C++
System-CoDesigner [10]	Actor-oriented Model in SystemC	Graph of all possible components	Mapping constraints in unspecified formalism	Yes	No	Yes	HW/SW partitioning, Task Mapping. Metrics: Performance, area	Perf. with VPC	Yes	Bitstream	Adapted C++
Advanced Systembuilder [11]	Functional description (process + channel)	HW components + topology	Mapping constraints	No	No	N/A	N/A	FPGA prototyping: HW profiler / CABA files generation	Yes	HLS of HW components	C code + API for communication
hArtes [12]	C code / Scilab language / Nu-Tech formalism	Architecture template in XML	Pragma annotations	Yes	No	No	Task Mapping, Task Parallelism, HW/SW partitioning. Metrics: Performance, area	SocLib performance estimation	Yes	HLS of HW components with Dwarv [13]	Compiled binary
PeaCE [14]	SPDF / fFSM / Task level models of computation	Set of available HW components	Specified through a GUI	Yes	No	No	Task Mapping, HW/SW partitioning, Scheduling. Metrics: performance, area, power	Cosimulation / FPGA prototyping	Yes, simulation + implementation	Synthesizable architecture	C code + comm
Space Codesign [15]	C/C++ code split into tasks	XML Specification	Specified through a GUI	No	No	N/A	N/A	Performance (TLM Simul.), Cost, power	Yes (with extra tools)	With extra tool (SpaceStudio GenX)	C/C++ adapted code
SYLVA [16]	SDF graphs	Set of Precharacterized HW components (FIMP)	Sampling Rate, Latency	No	No	Yes	Task Parallelism, Scheduling, HW configuration. Metrics: Performance, area, power	Area, Energy, Time	Yes	RTL HDL	N/A
Heracles [17]	Single or multi-threaded C/C++ code	Set of Parameterized HW components	Given through a GUI	No	No	N/A	N/A	Area, Performance	Yes	RTL HDL	Compiled binaries

Table 1: Comparison of existing ESL frameworks.

2.1.1 Inputs

We consider three types of inputs: the specification of the application, the hardware description, and constraints on the system (available resources, targeted performances, etc.). Ideally, the application should be specified with any available formalism or language that designers are familiar with. As shown in Table 1, in many ESL frameworks these functional descriptions are expressed with a Domain Specific Language, or as software (SW) code. This code must comply with some constraints, and usually must follow a model of computation — such as Kahn Process Network (KPN, [18])[9] or Synchronous DataFlow (SDF) [16]. These restrictions are necessary in order to enable the analysis and transformations (e.g. synthesis, optimization, etc.) required to produce the final system.

Most ESL tools also rely on architecture specifications. For instance, Space Codesign [15] uses XML descriptions of components that specify the inputs. Heracles [17] is another system that provides parameterized component templates that can be configured through a GUI. However, unlike our approach, both Space Codesign and Heracles do not perform automated DSE and consequently, designers have to fully specify the entire system before its performance can be evaluated. For TBES, the level of details of the specifications is chosen by designers through selected templates, and unspecified characteristics are automatically explored by our tool.

2.1.2 Design Space Exploration

The DSE should be fast even for large design spaces, by using efficient strategies to provide results satisfying the designers' constraints. To achieve this goal, the DSE process must rely on fast and accurate estimation techniques that will evaluate designs according to their performances, logic resource costs, and power consumption. ESL frameworks such as Advanced Systembuilder [11] and Space Codesign [15] do not perform a fully automated DSE. They rely on the designer for specifying the complete hardware architecture in order to produce synthesizable results. System Codesigner [10] uses a sophisticated but slow and non-deterministic multi-objective evolutionary algorithm for DSE. HLS is used but only in a first step to provide a hardware implementation for each actor selected by the designer. In SYLVA [16], an interesting automated DSE is proposed, starting from a SDF graph and a library of various functions implemented in hardware. It includes task level parallelism, hardware selection and pipelining. However, the architecture model is implicit and exclusively hardware-oriented.

In addition to the template-based strategy, another contribution of TBES is the integration of hardware accelerators exploration in the DSE loop. This is performed by generating series of accelerators by means of a fast HLS with different constraints. This estimation of IP candidates, before logic synthesis, allows to further improve the tradeoff between performance and logic resources utilization (cf. section 4.3.1). Automatic memory exploration and mapping is another important contribution of our work. This tedious task is not automated in most of the existing ESL tools

Finally, DSE must be scalable, i.e. designers should be able to balance the speed of the exploration — how long it will take to get a result satisfying the constraints — with the optimality of the results. To do so, our framework provides parameters that can be set by designers to define the DSE effort according to the desired tradeoff between optimality and solving time (cf. Section 4.4.1).

2.1.3 Outputs

The expected output of an ESL framework should be the complete system implementation artifacts: a synthesizable version of the hardware architecture code, along with a version of the software code adapted to the target architecture. In addition, our framework generates the project files and scripts for backend tools corresponding to the selected target (e.g. Xilinx FPGA). Several tools exist that generate a complete implementation of HMPSoC from the designer's specifications: [19] also generates an implementation ready for backend tools along with the adapted software tasks using a thread-based operating system to hide processor heterogeneity to the designer. LegUp [20] generates an implementation of the software and hardware architecture, and in addition performs an HLS of the tasks that the designer wants to accelerate, but without performing any exploration unlike our approach.

The solution introduced in our framework is based on Model-Driven Engineering (MDE, [21]). We use MDE, which advocates the specification of the system with a set of models to support analysis such that system capabilities and quality attributes (performance, reliability, etc.) can be predicted to discover problems before system integration and testing, to avoid costly rework later in the development process. Furthermore, the models provide essential support for design space exploration and code generation. We use the Architecture Analysis and Design Language (AADL, [22]) as main specification language, coupled with our architecture template strategy, to generate the final design. This abstraction level provides the flexibility to use different templates or modify existing templates to meet application specific needs.

2.1.4 Conclusion

Existing tools for HMPSoC design are all missing at least one of the features we think are essential for an industry-ready ESL framework. Either they do not perform automated DSE, like Advanced Systembuilder and

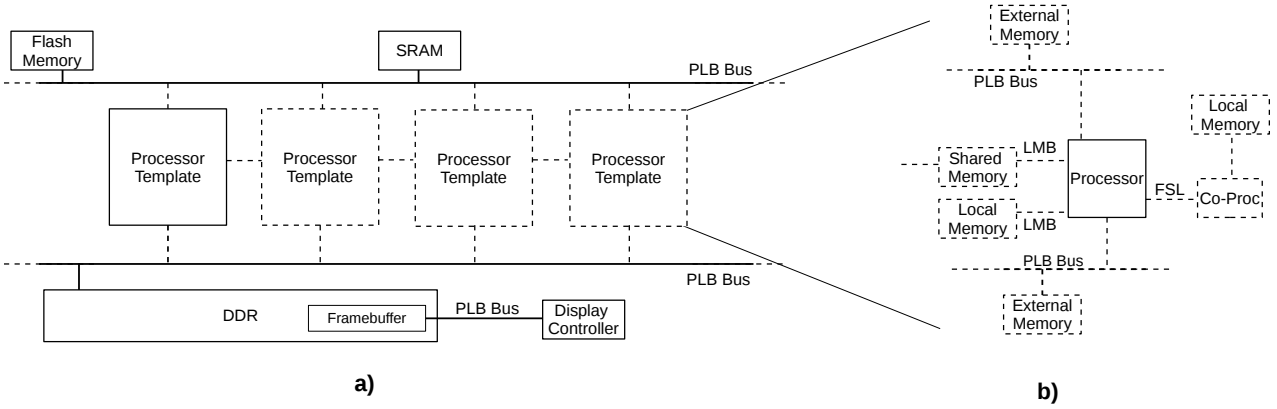


Figure 2: A graphical representation of an architecture template and a detailed view of the processor template. Solid lines indicate fixed parts (level 1) of the architecture while dotted lines represent potential instantiations (levels 2 and 3) that are decided during DSE.

Space Codesign, or their DSE process is too time-consuming — such as the SystemCoDesigner's CABA-based performance estimation, or the exhaustive exploration of Daedalus [9]. The hArtes framework [12] is too specific as it targets only a specific model of architecture: the MOLEN paradigm [23]. Consequently, there is a need for a new approach to tackle these issues, by providing designers with a way to express their specifications at the appropriate level of details, while automating the tedious tasks of the design flow.

2.2 Contributions

The contributions of this work are:

- An approach based on architecture templates that:
  - uses multi-level specifications providing flexibility, and usable by designers with different levels of expertise;
  - formalizes and bounds the search space, allowing to develop fast and accurate algorithms for power, performance and cost estimations and design space exploration;
  - automates code and script generation, simplifies testbed generation, which is a key aspect of the design process, and allows to extract, from DSE, solutions specified with a format which is compliant with backend implementation tools;
  - provides a solution to the lack of underlying platform architecture on FPGA, with a predefined but flexible architecture. This model, along with the encompassing standard Application Programming Interfaces (API) and Inputs/Outputs (I/O) information, and furthermore, libraries of software and hardware functions that can be easily updated, are of utmost importance, since they foster reuse.
  - simplifies, through the use of MDE, the adaptation of developed tools to new FPGA devices and vendor tools.
- The integration of the hardware IP exploration in the DSE through the use of HLS and fast and accurate model-based estimations.
- A DSE strategy that automatically explores relevant dimensions: processor allocation, hardware accelerator choice, task mapping, memory selection and data mapping, HW/SW partitioning and the degree of parallelism. The exploration relies on a highly scalable strategy that allows for a wide range of tradeoffs between the exploration time and the optimality of the solutions.

3 Template-based architecture specification

3.1 Template definition

Our model-driven approach relies on templates that provide pre-parameterized designs. The choice of templates is made by the designer in accordance with the application domain (DSP, video, etc.). Since for a specific domain, it is typical that different designs often have similar parameters, capturing these common parameters in a template favors reusability and reliability.

Since HMPSoC design is a complex process and requires a lot of competences, it is usually the case that different persons are responsible for developing the system: hence there are *domain experts*, that are responsible for developing the application aspects (code and task graph) and *optimization experts* that are responsible for

the implementation aspects (task mapping and data mapping). Our objective is to make our tool satisfactory for both purposes. To this end and in order to let designers focus their expertise on high added value tasks, we have divided our templates into three levels of specification. The first (*static*) level represents domain-specific elements that are fixed and cannot be modified during DSE. This is what is represented in solid lines in Fig. 2. The goal is to make the tool accessible to people with limited experience in design (e.g. *domain experts*). This is achieved by providing a library of templates — that could have been written by *optimization experts* — for specific application domains and device targets (e.g. Altera or Xilinx soft or hardcores). For the purpose of this study, we have targeted Xilinx and MicroBlaze (MB) architectures, but we used MDE methods to ease the management of libraries that can evolve through the integration of new templates and hardware targets.

The second level (*DSE bounds*) provides a place for designers to specify the DSE constraints that will bound the design space such as:

- a choice of architecture templates;
- minimum and maximum number of processors as well as the available types;
- available memories including their maximum sizes and available interconnects;
- a mapping of the application input and output data;
- a mapping of specific task(s) (input, output tasks typically);
- specification of application tasks that are candidates for hardware acceleration and/or duplication for data parallelism exploitation.

The selection of a subset of tasks candidates for hardware implementation and for duplication also relies on designer's experience. Just like for architecture templates, application designers can base this selection on their own knowledge of the algorithms and on profiling results. This information is used to constrain Cost/Delay exploration, which is based on HLS iterations and task nodes duplications. Moreover, to further simplify this step, some bounds directly related to the target platform are pre-defined. These are for instance, the number and types of logic resources (e.g. number of flip-flops, registers, memories, etc.) of a FPGA board.

The third level (*expert*) of specification targets all the decisions that can be taken by the framework during DSE. This level of specification is optional and can be used by experienced designers (e.g. *optimization experts*) to enforce the value of some attributes of the specifications that should not be changed during DSE. These attributes include:

- task and memory mappings;
- number and types of processors;
- hardware accelerators exploration, synthesis and integration with software calls;
- tasks duplication for data-parallelism exploitation;
- scheduling.

Templates must be created by first specifying the fixed parts of the system, corresponding to the first level of specification. It is possible to use templates of components such as processors, memories or external peripherals of the system. An example of a processor template is given in Fig.2-b. These templates can be stored in a library for future reuse, and to further reduce the design workload. Ideally, a public database could be provided where designers could release templates (or component templates) under a reusable licence, in a similar fashion to what is done for hardware IPs released with OpenCores [24]. Once the designer has specified the parts of the template that need to be completed (level two and possibly level three), the template can be used as input for DSE. Providing these levels of system specification can greatly simplify the design phase. Designers having at their disposal a template database only have to deal with the level 2 of the template specifications, and rely on automated DSE to optimize the details of level 3.

A graphical representation of an architecture template for a video-decoder is presented in Fig.2-a. In this example, the input (e.g. read a video file on a flash drive) and the output (e.g. write to the frame buffer) will most likely be the same from one design of the decoder to another. Hence the template pre-instantiates, in the architecture, the components that are necessary for the input and output operations. In addition, a video-decoder must be able to write the decoded output video in real-time. Since the data is in a raw uncompressed format, a large bandwidth is required and it is thus required to instantiate a dedicated bus to the frame buffer.

```
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_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 3: Excerpt of the AADL model of the MicroBlaze.

3.2 AADL Specification

To model the system architecture, AADL [22] is used. It is a domain-specific language for embedded systems including modeling capabilities for both software and hardware parts of a system. AADL provides base component categories for representing processors, buses, memories, devices, systems, processes, threads, thread groups, data and subprograms. In TBES however, the software part is currently specified using KPN-compliant C code, and only the hardware platform is specified in AADL.

A component in AADL is described using two distinct types of declaration: i) a *type* declaration specifies the external interface of a component through which it can be connected to other components (I/O ports, data/bus access, etc.); ii) an *implementation* declaration specifies the internal composition of a component in terms of one or several subcomponents. Both *type* and *implementation* declarations can specify component properties. A set of properties predefined in the AADL standard is available to specify various configuration parameters of components. For example, for a generic memory component, the predefined properties are the size, the access rights, the word size, read and write times, etc. AADL can be extended by the addition of other properties for representing attributes of more specific components such as a MB processor. This is illustrated in Fig.3, where an excerpt of the AADL representation of a MB processor is presented.

AADL components and property sets declarations can constitute a library of components to be instantiated in FPGA-based designs. Declarations for Xilinx-specific components such as MB processors, buses, controllers, etc. have been added to a library. This is necessary to specify the values for parameters specific to Xilinx IPs.

From AADL models, we are able to generate all the implementation files for the found architecture solution needed by synthesis and implementation tools such as Xilinx XPS or Altera's Quartus. The Xilinx tools have been selected to demonstrate the proposed approach, so the generated files are the *.mhs* and *.mss* files [25]. The first file describes the synthesized hardware platform with its components, their parameters and their connections. The second file describes the driver specifications in order to call the hardware components from the software code.

The architecture selected from DSE is represented by an AADL model, which contains the instantiated components, their associated parameters represented as properties, and the components' inter-connections. This representation is automatically transformed, into the corresponding project files for the targeted FPGA design tools for an immediate implementation on the FPGA. Using this MDE technique, only the grammar and the model transformations used for code generation need to be changed to target a different FPGA synthesis tool.

4 Design Flow

4.1 Tool Philosophy

While it is now clear that efficient and easy-to-use tools are a key success-maker for embedded systems development, the impact of ESL tools has been underestimated for years. A layered approach has emerged as a divide-and-conquer strategy to cope with the increasing complexity of programming such platforms. Tools are critical in the design process – as the tools serve as the user's entry point, and directly impact the final system's performances –, it is thus essential to identify the characteristics that make a tool efficient, usable, and acceptable to users. The first challenge that tool makers must face is to offer an environment that matches the designer's habits, thus reducing the learning curve in order to lower the rejection rate from experienced – hence highly valuable – users. The tools do not aim at replacing the user's knowledge. However, a significant gain in development time is expected by having the tools handle the tedious tasks, while designers concentrate their efforts on specialized tasks requiring expertise. As a consequence, the tools must preserve the legacy artifacts

(process, IPs, etc.), since this provides continuity to the users, flattens the learning curve, and reduces the time-to-market.

From an internal point of view – being relative to tool designers – new techniques must be applied to cut off the development and testing costs of software (MDE, code generation, etc.). This is enabled by frameworks that capture the intent of existing tools as well as their API. A direct benefit lies in the ability to tailor the execution of these tools, through scripting, function invocation, etc. This is a critical issue: it leads to gaining a short term solution by implementing tailored or customized flows controlling the execution of external tools, which can be used as a comparison point to evaluate every potential future evolutions of the flow. Candidate tools to be integrated in the flow require no more than one useful functionality (transcoding, etc.) to be kept in the flow. Because their internal flow is exposed as atomic steps, elementary functions drive the selection process. Making the right choice, though, requires an educated guess coming from a deep knowledge of the domain. While being based on existing tool integration, such approaches also favor evolution.

4.2 The TBES Tool

Following the presented tool philosophy, TBES was constructed by assembling different tools — already existing tools and custom-made processes — into a flow depicted in Fig.7. The next subsections present each step of the flow with their associated tools.

4.2.1 Input Specifications

The required inputs for our framework are the C code of the application and a template describing a generic architecture that will be used as a basis for DSE.

For the C code input, designers must split their application into tasks, in order to express the application parallelism. This must be done manually as task splitting is application-specific and will also determine the granularity of the communication. Deciding the task granularity using automated methods is very challenging, whereas it is natural for a designer who has the knowledge of the application. To help the designer, our framework includes an automated profiling process based on the gprof standard tool[26].

Once the task splitting is done, the C code of the application is automatically transformed to comply with the Kahn Process Network (KPN, [18]) model of computation. Each node of the resulting KPN graph represents one task of the application, and the edges represent the communication channels. The transformation is performed by a tool called KPNGen [27], which requires that the input application satisfies a number of constraints: the program must be coded in the form of one or more nested loops or conditional statements, wherein functions are called sequentially, as illustrated by Fig.4. In the loop(s), each function call corresponds to a single task,

```
for (i = 0; i < 1138; i++) {
   for (j = 0; j < 42; j++) {
          fetchTask(&iqzz_d);
          iqzzTask(&iqzz_d, &block_YCbCr);
          idctTask(&block_YCbCr, &Idct_YCbC);
          yuvTask(&Idct_YCbC, &pix);
          dispatchTask(&pix);
   }
}
```

Figure 4: Example of the main loop of a program compliant with the constraints imposed by the KPNGen tool.

and communications are expressed through the call parameters, i.e. a variable used by two functions means that this variable is sent by one function to another. Other restrictions that apply on the program are that the indices of the loops must evolve following affine functions, and the control of these indices must be static, i.e. they must be determined at design time and cannot be modified during execution. The advantage of using KPN is that it makes the execution of the program deterministic, and the task synchronization is performed through blocking reads, thus allowing to easily exploit data parallelism.

For the architecture input, it is specified through the use of templates expressed in the AADL language as introduced in subsection 3.2.

4.3 Performance and Cost Estimations

Two properties are estimated during our DSE targeting FPGAs: resource usage and performance. The efficiency of DSE must rely on accurate estimations. In this section, we describe these estimations performed at several stages of our design flow.

4.3.1 HLS-based Coprocessor Exploration and Estimation

We use HLS to perform an exploration of different coprocessor solutions inside the loop of the heterogeneous multiprocessor DSE. This exploration is used to find the best compromise between performance and logic

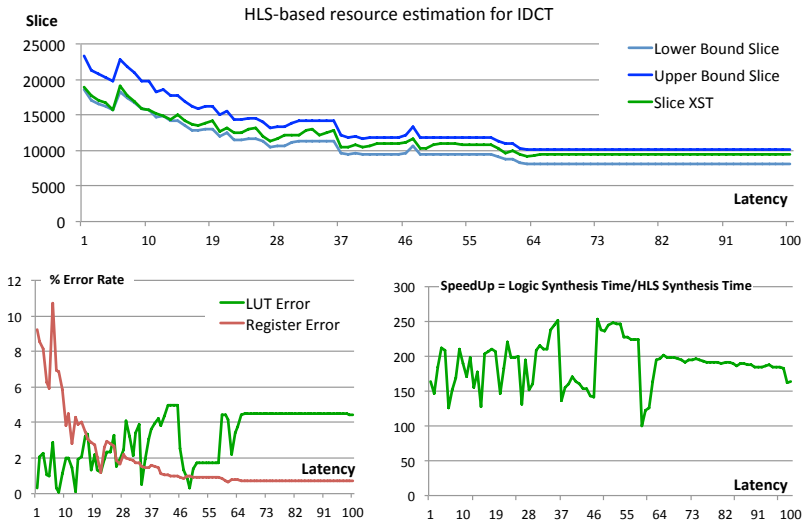


Figure 5: Exploration of an IDCT coprocessor in order to compare our HLS-based estimation with logic synthesis. Comparisons of slice estimations are shown on the top curve and the bottom ones show the accuracy of the estimation and the speedup over logic synthesis.

resource consumption. This is made possible by coupling HLS with accurate resource consumption estimators, instead of performing a costly logic synthesis of the coprocessors.

HLS is the transformation of a functional description — provided as C code in our framework — into a Register Transfer Level (RTL) description. The generated architecture described at the RTL level is compliant with an architecture model that depends on the HLS tool. The HLS tool currently used in our framework is GAUT [28], which is linear in complexity. It is a free HLS tool based on a Data Flow Graph model and therefore dedicated to data-dominated algorithms. From a C/C++ specification and a set of design constraints, GAUT automatically generates a potentially pipelined RTL architecture in two formalisms: in VHDL for synthesis, and in SystemC for virtual prototyping. In our flow, the set of design constraints is provided by the HLS/DSE controller (cf. Fig.7), in order to generate and evaluate a set of coprocessors with several performance/resource tradeoffs. The HLS constraints we consider for automated exploration are the latency and the communication model (FIFO or shared memory). The generated architecture is composed of i) a processing unit composed of the data-path (logic/arithmetic operators + storage elements + steering logic) and an FSM controller; ii) a memory unit composed of memory banks and associated controllers and iii) a communication interface which can be implemented as a FIFO, a shared memory (optionally with ping pong mode), or a 4-phase handshake. Given that the underlying model of architecture of GAUT is clearly defined, it is possible to perform a resource usage estimation. So after behavioral synthesis, the following features are known exactly:

- allocated data-path logic/arithmetic operators: numbers of DSP blocks or LUTs;
- number of registers;
- width and height of the FSM controller: slices number (Distributed RAM) or BRAMs;
- number and size of multiplexers;
- width and size of memories: slices number (Distributed RAM) or BRAMs.

From these features, we can estimate the usage of FPGA resources: LUTs, registers, DSP blocks, block RAM, etc. Fig.5 presents the results of resource estimation on an IDCT algorithm from the MJPEG benchmark. For this experiment, we generated a series of IDCT coprocessors with varying latency, and then from the HLS results computed an estimation. We then performed a logic synthesis with the Xilinx XST synthesis tool to measure the accuracy of our estimation. For slices (which is the basic logic block in Xilinx FPGAs, gathering LUTs + registers + dedicated multiplexer), the tool computes a lower (100% use of slices resources) and an upper bound (80% use of slices resources). The upper bound rate has been set (80%) from HLS benchmarks in the domain of signal and image processing: FFT, IDCT, Sobel, Gaussian filter, Walsh-Hadamard transform, FIR, IIR, Cordic, AES, etc. The results of these benchmarks and more details on our estimation method can be found in [29]. The bottom left graph of Fig.5 shows, for the same IDCT series, the error percentage between the number of LUTs and registers obtained with our estimation and the real cost obtained after logic synthesis. We can see that the estimation error of LUTs is usually under 5% and the estimation error of registers is higher (up to 10%) in some cases, probably due to fan-out sizing reasons (register duplication). The bottom right graph shows that we obtain speedups of two orders of magnitude over logic synthesis. Moreover, the speedup increases with application complexity and allows to quickly explore the design space of solutions.

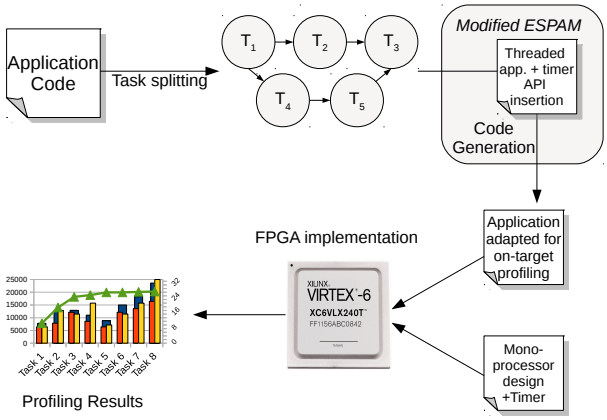


Figure 6: Flow of the on-target profiling step.

In conclusion, HLS offers a fast implementation solution and a possibility to explore the design space of hardware IP in an efficient way. A quick and accurate estimation of results is useful because designers can quickly measure the effects of code transformation and resource and timings constraints.

4.3.2 Application Profiling

The exploration starts with an automated and non-intrusive on-target profiling, which is performed to obtain, for each task, accurate performance measurements for the targeted processor. These measurements are then used during DSE for mapping performance estimation. They also provide a way to determine a set of task candidates for hardware acceleration.

In our framework, this profiling is automated through code transformation and the use of a pre-designed FPGA system that contains the target processor (e.g. MB) with hardware timers to measure execution time. That step is illustrated by Fig.6. During the code transformation, which is performed by a modified version of the ESPAM tool [30], each task becomes a thread and calls to the timers are inserted. Inter-task communication times are not measured during profiling as they depend on design decisions such as task and data mapping, which have not yet been taken at this stage. During this profiling, it is the responsibility of the designer to provide a dataset representative of the final system execution.

4.3.3 Multiprocessor without Interconnect and Dependencies

During early stages of the DSE process, when only the number and the type of processors have been decided, a first estimation is performed. We first check that the design does not consume more logic resources than what has been specified. If this is the case, the design is discarded and not further explored. Otherwise, we perform a performance estimation. At this stage, such estimation is obviously very rough. However, it is only used to establish if acceleration is needed. The estimation is given by the following formula:

$$\frac{CT}{nP} + nW \times nWC + nR \times CC \tag{1}$$

where CT is the computing time, nP the number of processors, nW the number of write operations, nWC the cost in cycles of a write operation, nR the number of read operations, and CC the cost in cycles of a cache miss. In this equation, we divide the total execution time of the application — computed using the profiling described in the above section — by the number of processors, which yields the maximum possible acceleration. While this is overly optimistic, it is counterbalanced by the communication cost for which we assume that a cache miss occurs at each read.

4.3.4 Communication

Since dataflow applications are usually data-intensive, design choices for communication such as data mappings, memory types and sizes, bus types are as equally important as computation design choices. So in order to detect possible contentions on a bus or a memory, we first compute the number of cycles necessary to read — or write since the formula is the same — the quantity of data for one iteration of the program from a given memory. It is computed from the following formula:

$$nRC = RL_{mem} \times \frac{dataSize}{B_{mem}} + dataSize \tag{2}$$

where nRC is the total number of cycles used for read operations in one iteration of the program, RL_{mem} the latency value in cycles for a read operation in the memory mem, and B_{mem} the number of bytes transferred

in one burst. With this information, we can estimate the risk of congestion as:

$$CR_{mem} = \frac{F_{mem}}{nI \times (nRC + nWC)} \tag{3}$$

where CR_{mem} is the congestion ratio of the given memory mem, F_{mem} the frequency of the memory, nI the number of iterations of the program in one second, and nRC and nWC the number of cycles for the read operations — write operations respectively — given by the previous equation (2). The result is a ratio, which when higher than one means that the memory is overloaded. When lower than one, then no congestion should occur. A similar formula can be used for communication channels:

$$CR_{comm} = \frac{F}{L_{comm} \times (dataSize/B_{comm})} \tag{4}$$

where CR_{comm} is the congestion ratio of the given communication channel $comm$, F the frequency of the system, L_{comm} the latency of the communication channel, and B_{comm} the bandwidth of the channel in bytes per second. Both of these metrics are used during our data mapping exploration in order to estimate the degree of congestion of a memory or a channel.

4.3.5 Simulation

Once the full system has been specified, a final performance estimation is performed with the Sesame simulation tool [31]. This simulation is trace-based and takes into account three kinds of events: *execute*, *read* and *write*. Using the trace of these events and a description of the architecture and its components, Sesame provides a fast estimation of the system performances. While being significantly faster than a Cycle Accurate-Bit Accurate simulation, it is still the most time-consuming operation of our DSE flow. The total simulation time depends on the size of the trace and the number of designs to evaluate. It is thus important on one hand to limit the size of the trace by finding the smallest execution time that still produces results representative of the final performance, and on the other hand to perform a selection of the designs to be simulated. The version of Sesame implemented in the Daedalus framework is used to generate exhaustively the task mappings. We thus modified it in order to take into account only the mappings selected by our tool, leading to shorter exploration times as shown in the results section.

4.4 Design Space Exploration

The DSE flow is shown in Fig.7, and a more detailed description of the DSE algorithm can be found in [29]. We focus here on the task and data mapping, as well as on the pruning strategies used to provide designers with the possibility to balance exploration time and solution optimality.

4.4.1 Pruning Strategy

Our goal is to offer to designers a highly scalable exploration strategy, which ranges from a greedy algorithm that will return the first solution that satisfies the constraints, to an exhaustive exploration, and includes in between a very large number of intermediary solutions that offer a tradeoff between the exploration time and the optimality of the resulting solution. To this end, designers can specify through variables (the N_1, N_2, N_3 in Fig.7), the maximal number of solutions they want to keep at the end of each step of the design flow. Consequently, one can decide to keep a larger number of solutions on the steps that would need more exploration. This selection process is shown in Fig.7 and in Algorithm 1. The selection relies on metrics used to evaluate the performance or the cost of the system such as the ones described in Section 4.3.

4.4.2 Task and Data Mapping Heuristics

The algorithm for data and task mapping is described in Algorithm 1. A few definitions must be introduced to understand the algorithm:

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

The data mapping consists of assigning the data communicated between the tasks onto the memories of the architecture. 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. Since we want to maximize the size 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 produce interesting variations:

Algorithm 1 Data and Task Mapping Algorithm.

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

for all N_1 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_2 best data-mappings

  //**Task Mapping**
  for all N_2 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_3 best task-mappings
end for

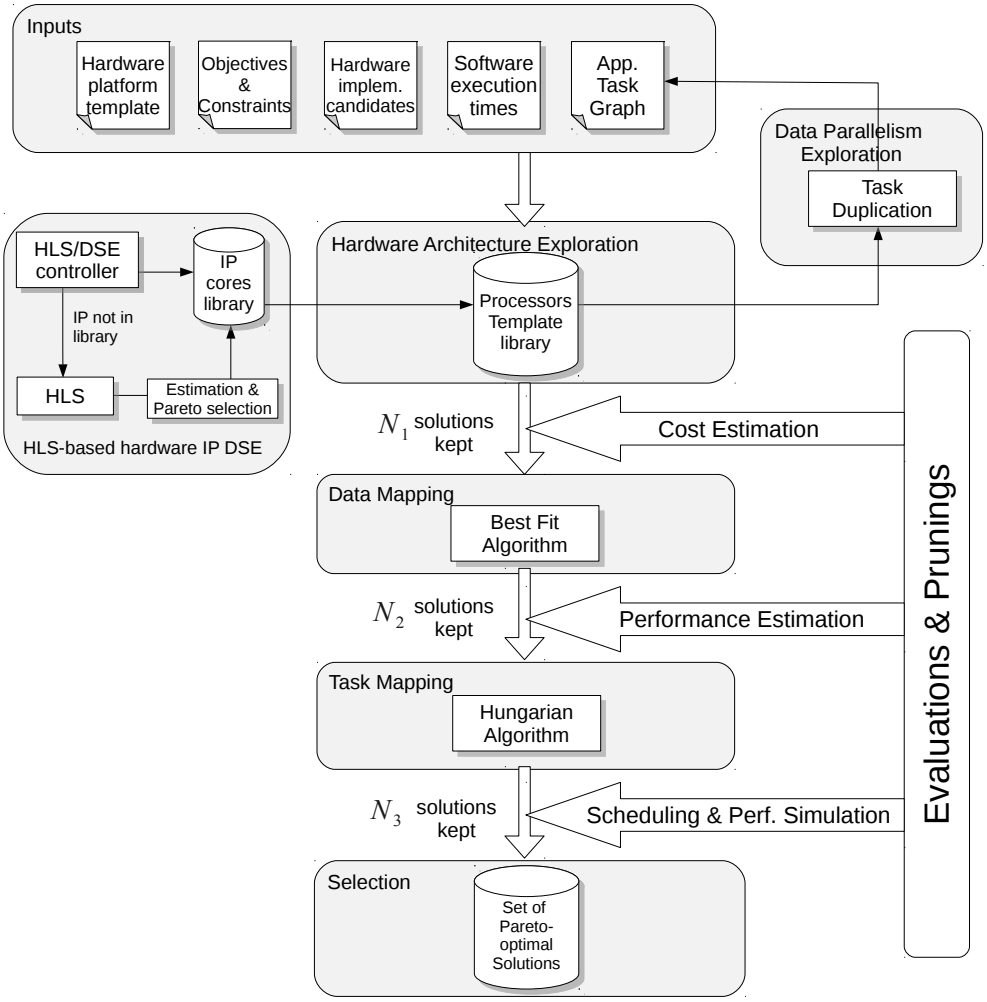


Figure 7: Flow of the Design Space Exploration.

- The read and write latencies of the different memory types are modified with a random coefficient that introduces in the DSE variations representing bus access conflicts and cache misses.
- Memories that are synthesized, such as BRAM, do not have a fixed size. So another dimension explored during data mapping is memory size. The allowed sizes must be provided by the designer.
- Last, two ways for the order in which data are mapped are considered: biggest data size mapped first and smallest data size mapped first.

Following these strategies and using the performance estimation techniques described in 4.3.4, several data mappings are generated. Potential duplicate mappings are removed from the set of data mappings generated following these rules. The pseudo-code of the algorithm can be seen in the upper part of Algorithm 1.

To perform the task mapping, we used a method based on the Hungarian algorithm [32], which provides an optimal solution to the assignment problem in polynomial time. The pseudo-code of the task mapping is described in the lower part of Algorithm 1.

In our implementation, the cost matrix for the Hungarian algorithm represents the cost of assigning a task of the current task cluster to one of the available processors. Each cost ω is computed using the following formula:

$$\omega = (\alpha \times ET_{T,P} + \beta \times C_{T,M})(\gamma \times PL_P)(\delta \times ML_M) \tag{5}$$

where 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. Execution and communication times are used to determine the performance of a task mapped on that unit. $ET_{T,P}$ is the number of cycles needed by the processor P to compute one iteration of the task T. $C_{T,M}$ 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 performances: 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. PL_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. ML_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 determined, the "synthesizable" memories, i.e. memories which are synthesized and for which the size is not static (e.g. BRAM), must be assigned to a processor. For each piece of data that was mapped on such memories during the prior data mapping step, we check if it corresponds to a communication between two tasks that are mapped on the same processor. In such case, the memory is implemented as a local memory of the processor (as seen in Fig.2-b). Otherwise the memory 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.

4.5 SoC Production

Code generation is performed following a model-based paradigm, i.e. as a chain of transformations of models representing every specification involved in the design flow. Model transformations have been developed using the ATL language [33]. From a set of AADL specifications, models are generated to be taken as input by the synthesis tools for the specific FPGA platform (e.g. the *.mhs* and *.mss* files for Xilinx). We use the Xtext tool [34], which is an Eclipse-based tool to simplify the support of Domain Specific Languages (DSL) by generating several tools (parsers, serializers, code generators, etc.) from the specifications (grammar, properties, etc.) of a DSL. So, grammars have been developed for the *.mhs* and *.mss* formats, and the Xtext tool has been used to automatically generate parsers and serializers allowing to convert these files into models. It ensures that the generated code is correct through conformance to the grammar/meta-models. These models can also be handled by the numerous model-driven development tools, which greatly help in the development of the tool chain.

4.6 Limits of the Current Target Architecture

With the current version of TBES, the architecture must contain at least one microprocessor, which can be linked to one or several peripherals (memory, I/O, sensor, etc.) through one or several links (shared or dedicated). Each processor can also have one or several coprocessors. However, those coprocessors cannot yet be cascaded. The number of processors is limited by the bus-based communication. Templates based on Network-on-Chips are a solution to these problems. However, we may need to import a different analytical model from the literature since the current DSE algorithm is based on a bus communication model. In this work, we specifically targeted FPGAs that were used to validate our approach. However there is no theoretical obstacle to target ASICs provided that models of specification and scripts for the backend tools can be developed.

5 Results

Various benchmarks have been performed during the development of our DSE algorithms and tools. In this section, we present two applications that demonstrate the efficiency of our solution. First, a standard MJPEG decoder [35] is used as a well-known benchmark. Then, a complex Viola-Jones face detection application [36] is presented to better demonstrate the capabilities of the framework.

5.1 MJPEG

After the on-target profiling (as described in Section 4.3.2), the MJPEG decoder was split into five sequential tasks: *Decode*, *IQZZ*, *IDCT*, *YUV* and *Display*. The movie used as example has a resolution of 256×144 pixels. The template used for the exploration is shown in Fig.2-a. Since the target board is a Xilinx XUPV5 FPGA board [37], components specific to Xilinx are instantiated in the template. In this figure, elements specific to this application are drawn with solid lines, so the architecture contains 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 line elements represent components that can be instantiated — e.g. there can be up to four processors instantiated in the design. Fig.2-b illustrates the details of the processor template. It shows that a processor can be linked to a local memory or a shared memory through a LMB bus. It can be linked to one or more external memories through a PLB bus and can also be linked through an FSL to a coprocessor, which can itself have a dedicated local memory. These architectural elements represent the static part of the template. The rest of the settings for the DSE-bound level are:

- the number of processors, which is between one and four with all of them being of MB type;
- the available memories, which are the DDR, BRAM and SRAM;
- the hardware accelerators for IDCT and YUV;
- the IDCT can be duplicated to exploit data-parallelism;
- the performance objective, which is set to 24 frames per second (FPS);

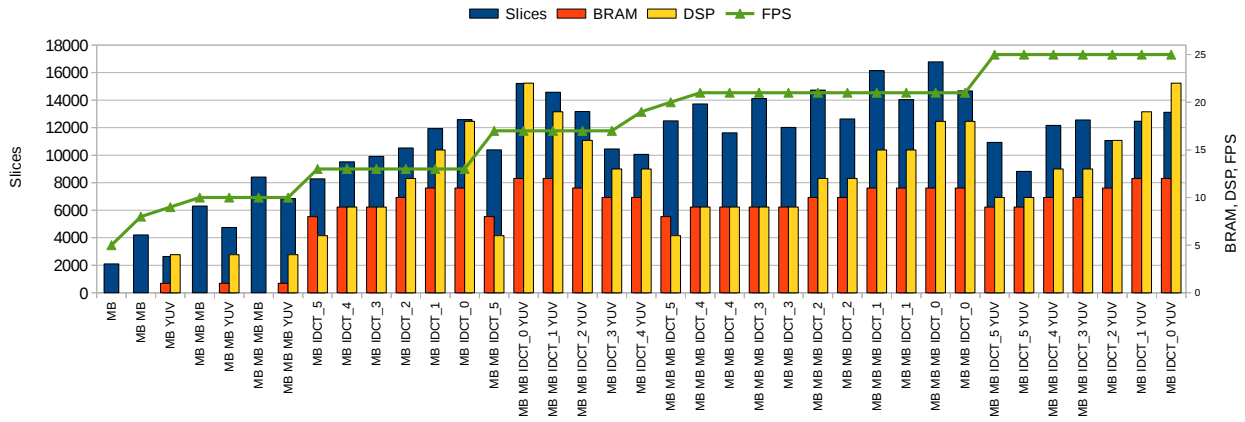


Figure 8: Results of the exploration for the MJPEG by our framework sorted by increasing FPS.

– the N_i variables that set the scalability of the DSE (see Fig.7), which were set to -1, so that no pruning was performed.

Fig.8 shows a selection of results for the DSE. In these results, $IDCT_X$ designates a MB with an accelerator for the IDCT task; the X shows the different versions of the accelerator — the smaller the X, the faster the IP (i.e. the IP has a lower latency). The same holds for YUV. If we exclude the time for the hardware accelerators exploration (about 5 minutes) — by assuming for instance that the IPs were already present in the database —, the total time taken to produce these 37 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 reached our objective of 24 FPS.

In order to evaluate the efficiency of the Hungarian algorithm-based method, we compared the best mappings found with the Hungarian method with the optimal mappings found with an exhaustive mapping exploration, as it is performed in the Daedalus framework [9]. The results are given in Table 2, where they are sorted in increasing order of performance according to the Sesame estimation of the exhaustive mappings. The mapping columns specify the task mapping onto the processors: there as many numbers as there are tasks — 5 in this case —, 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_5 is the processor 1 and YUV is the processor 2. 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 (*YUV* and *Display*) are mapped on YUV. The columns *Sesame Est.* are the numbers of cycles estimated by Sesame for the execution of the application.

We can see that in 11 cases out of 37, there are no or little differences in performances, meaning that our Hungarian method has found the optimal solution or a near-optimal one. On average, the difference in performances is 12%. This difference is relatively small considering the speedups obtained with the coprocessors in this case study. The important point is also the pruning ratio obtained with the Hungarian algorithm for this example. If the mapping generation was exhaustive, then over 200 000 architectures and mappings would have been generated. It means that with a relatively small MJPEG example, it would have taken over a day to evaluate all the solutions 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, with an average ratio of 550 for speedup per percent loss.

The next steps of the design process are the synthesis of the whole architecture and the execution on the target FPGA. Considering synthesis times of complex architectures and execution, electing the most relevant ones out of many is of major importance. Table 3 shows the comparison between our estimates and the implementation on FPGA over six architectures. We can see that our estimates are pretty accurate with an average error of 7.3%. It is also important to keep in mind that the estimation process is compliant with the tool approach targeting fast design so that multiple solutions can be seamlessly generated and tested.

5.2 Viola-Jones based Face Detection

Application and Template As an example of a complex application, we used a face detection software based on the Viola-Jones object detection method. This application was split into eleven tasks as illustrated in Fig.9. After loading the necessary data (tasks 1 and 2), the application performs a series of transformation on the picture (tasks 3 through 9) in order to increase the efficiency of the actual detection of faces (tasks 10 and 11). The generic architecture used for the exploration is similar to the one for the MJPEG (cf. Fig.2-a), except for the SRAM memory, which is removed since it is absent from the ML605 FPGA board [38] targeted for this application. The processor template, shown in Fig.2-b, is also similar to the MJPEG one. The *DSE-bound* settings given as parameters for the exploration were:

Table 2: Comparison between Hungarian-algorithm based method and exhaustive-search mapping exploration.

Architecture	Hungarian-Algo.		Exhaustive Search		Difference	Diff %
	Mapping	Sesame Est.	Mapping	Sesame Est.		
MB	0 0 0 0 0	718915412	0 0 0 0 0	718915412	0	0
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

Table 3: Comparison of FPGA implementations of MJPEG architectures and performance evaluations with TBES

Architecture	FPS	Est. FPS	Cycles count	Est. Cycles count	% Error	Frequency (MHz)
MB	5.32	5	681059750	718915412	5.56	125
YUV	5.71	5	634948450	639242454	0.7	125
IDCT_0	9.48	8	382378299	412204302	7.8	125
MB MB	9.95	8	364346144	422931756	16.08	125
MB YUV	10.43	9	347540716	393331640	13.18	125
MB IDCT_0	13.42	13	270104971	268459365	0.61	125

Table 4: List of the communication channels between tasks with the types and sizes of exchanged data.

Linked tasks	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
4-5,5-6,6-7,7-8,8-9	IMG_RESIZE_GRAY	65416
9-10,9-10	IMG_INTEGRAL	279056
9-10	SQUAREIMG_INTEGRAL	558080
10-11	RESULTS	2400

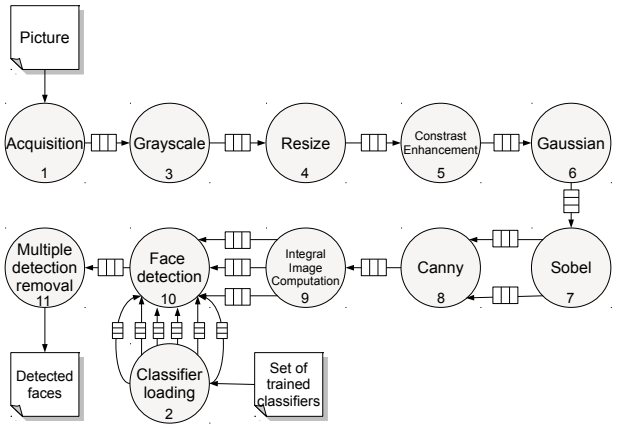


Figure 9: Face detection application split into 11 tasks with links between communicating tasks.

Table 5: Results of the automated mapping exploration.

Architecture	Task Mapping	Sesame Est.	Memories	Memory Mapping	Speedup
MB	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
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 0 0 1 1 1	1,2
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 0 1 1 1 1 1 1 1 1 1 1	1,65
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 0 0 1 1 1	1,69
HEQ	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	1,34
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 0 0 1 1 1	1,63
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	1,74
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	1,6
DET	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	1,81
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 1	2,23
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 1	2,34
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
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
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
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	1,9

- there can be between one to four MBs;
- there are two memory types available: DDR and BRAM;
- the *Face detection* task can be duplicated to exploit data-parallelism;
- the tasks *Contrast enhancement* and *Face detection*, respectively noted HEQ and DET in the results are specified by the designer as accelerable through hardware;
- the N_i variables that set the scalability of the DSE were all set to -1, meaning that no pruning was performed.

First results In order to evaluate the efficiency of our algorithm, we have launched an exploration with fully-automated memory mapping decisions and with hardware acceleration allowed. Instances of target accelerators were already present in the database and thus no HLS-based exploration was performed. Their characteristics are specified in Table 6. In this experiment, we consider applying face recognition on a video stream as a smartcam application. So we consider several iterations of the program by executing face recognition on a set of subsequent video frames coming from a camera. A consequence of having multiple iterations is that it reduces the impact of classifiers loading (the *Classifier loading* task), since this operation has to be performed only once for all the images. Moreover it also offers interesting inter-frame parallelism opportunities.

Table 5 shows the best results based on performances for the given hardware architecture. The first column shows the description of the architecture: MB DET HEQ means an architecture with three MB processors, DET is a MB with one DET hardware accelerator and HEQ is a MB with one HEQ hardware accelerator. The second column specifies the task mapping: they are described in a similar fashion as what was done for Table 2. Thus the mapping *2 2 0 1 2 0 0 0 0 1 2*, means that the first two tasks — tasks are in the same order as in Fig.9 — are mapped on HEQ, the third task is mapped on MB and so on. The memory mapping is specified as

Table 6: Hardware accelerators characteristics.

Accelerator Type	**HEQ**	**DET**
Speedup over SW	90	53
Number of Slices	9308	471
Number of DSP blocks	256	0

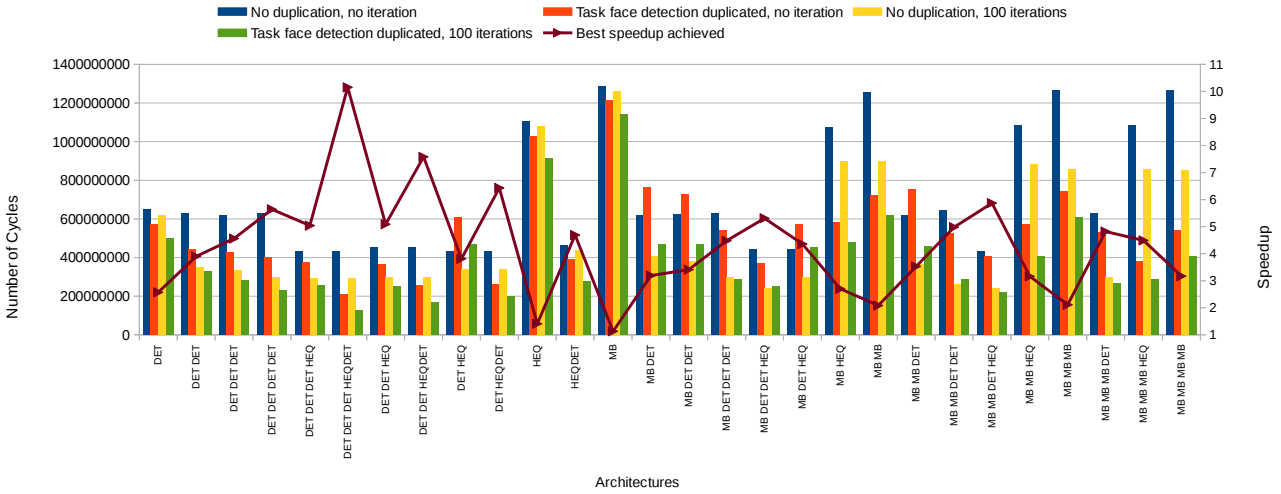


Figure 10: Viola-Jones face detection: DSE and results of parallelism exploration.

follows: there are as many numbers as there are communication channels in the application — the order is the same as that of Table 4 — and their value 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 is computed from the worst case found in these given selected results, which is the full software monoprocessor solution.

We can see in Table 5 that results with good performances are obtained 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 through the Sesame simulator in 230 seconds, resulting in a total exploration time of 234 seconds. This example shows the issue of scalability for large applications, for which exhaustive mapping exploration is no longer possible: for two processors, the task mapping would result in 2048 solutions, and with two memories and 18 communication channels, there would be 262 144 possible data mappings, resulting in a total of over 536 million solutions to evaluate, if the explorations for both data and task mappings were exhaustive.

Task Duplication & Pipelining Our environment aims at offering to designers the ability to explore data and task parallelism without spending time working on low level and error prone implementation details of the system. So another series of four experiments were performed in order to study the impact of parallelism at the application level. The first experiment is used as reference since no parallelism is exploited. In the second experiment, only data-parallelism is explored, while in the third, only task-parallelism is explored and in the fourth, both types of parallelism are explored. The exploration of data-parallelism was achieved by duplicating the *Face detection* task three times, since it is the most resource-consuming task. To explore the pipelining, we performed several iterations of the program, in a similar fashion to the previous experiment. The accelerators are the same as for 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 assumed that all communications would fit on the BRAM memories, which have a latency of one cycle.

A selection of results is presented in Fig.10. For architectures having more than one processor, we can observe that the duplication of the *Face detection* task provides better performances. However in some cases, the version with duplicated tasks is slower than the version with no duplicated task. This happens when the architecture contains at least one processor that does not have a coprocessor for the *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. Otherwise, our tool has a built-in rule that checks that there are enough coprocessors given the number of times a task was duplicated, and that discards results that do not verify this assumption. If we remove 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 at no additional resource costs.

Overall, the best architecture yields a speedup of 10.15 over the worst case, which is 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 0.79 image per second.

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

6 Conclusion

This paper presented a complete system level tool for the design of heterogeneous multiprocessor architectures. The design flow has been demonstrated using Xilinx FPGAs. Our solution relies on three strong points. Whereas designers of reconfigurable architectures traditionally spend a great deal of time and design effort for each design as reusability is missing, our template-based approach promotes reuse and cuts off design effort. Secondly, it provides a framework for estimation, DSE and code/script generation. Designers benefit from years of legacy and irreplaceable know-how. Not only does our solution automate tedious and error-prone design tasks, but it unleashes the productivity of application designers by putting their skills at the heart of the process (e.g. template selection, tasks identification for duplication, etc.). However, designer experience may not be sufficient to identify the proper design options. Our solution thus thirdly offers a fast and automatic DSE method, that includes HLS in the exploration loop, hardware accelerator selection, task and data mapping. It allows a significant reduction of the set of solutions to be simulated, and brings a significant speed-up with minor impact on the overall solution quality. Real cases including a complex face-detection application served as from-specification-to-implementation demonstrators.

New FPGA generations, such as the Altera Stratix 10 family [39], offer more logic resources and embed hardcore CPUs, and thus they further increase the design space and justify the need for scalability. We believe that our template-based approach, allowing our flow to adapt to new architectures, along with the adaptable scalabilty made possible with our approach, is a relevant solution to face the challenges of designing HMPSoC for these new platforms.

This work opens promising perspectives. At the application's capture level, we are working on extensions encompassing new models of computation beyond KPN. At a lower level, to complete the automation, MDE is considered as a smart way to generate codes and scripts for third party tools and devices such as Altera.

References

[1] Miriam Leeser, Shawn Miller, and Haiqian Yu. Smart camera based on reconfigurable hardware enables diverse real-time applications. In *Field-Programmable Custom Computing Machines, 2004. FCCM 2004. 12th Annual IEEE Symposium on*, pages 147–155. IEEE, 2004.

[2] Umar Alqasemi, Hai Li, Andres Aguirre, and Quing Zhu. Fpga-based reconfigurable processor for ultrafast interlaced ultrasound and photoacoustic imaging. *Ultrasonics, Ferroelectrics, and Frequency Control, IEEE Transactions on*, 59(7):1344–1353, 2012.

[3] Przemyslaw Pawelczak, Keith Nolan, Linda Doyle, Ser Wah Oh, and Danijela Cabric. Cognitive radio: Ten years of experimentation and development. *Communications Magazine, IEEE*, 49(3):90–100, 2011.

[4] D. Suzuki, N Natsui, A Mochizuki, S Miura, H. Honjo, K. Kinoshita, H. Sato, S. Ikeda, T. Endoh, H. Ohno, and T. Hanyu. Fabrication of a magnetic tunnel junction-based 240-tile nonvolatile field-programmable gate array chip skipping wasted write operations for greedy power-reduced logic applications. *IEICE Electronics Express*, 10(23), 2013.

[5] M. Sadri, C. Weis, N. Wehn, and L. Benini. Energy and performance exploration of accelerator coherency port using xilinx zynq. In *Proceedings of the 10th FPGAworld Conference*, page 5. ACM, 2013.

[6] Altera. Altera and IBM Unveil FPGA-accelerated POWER Systems with Coherent Shared Memory. `http://newsroom.altera.com/press-releases/nr-ibm-capi.htm`, 2015. Last accessed: 14/04/2015.

[7] K. Benkrid, D. Crookes, and A. Benkrid. Towards a general framework for FPGA based image processing using hardware skeletons. *Parallel Computing*, 28(7–8), 2002.

[8] Mark D Hill and Michael R Marty. Amdahl's law in the multicore era. *Computer*, (7):33–38, 2008.

[9] 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.

[10] 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 Trans. on Design Automation of Electronic Systems (TODAES)*, 14(1):1–23, 2009.

[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*, 2010.

[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.

[13] 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. Int. Conf. on*, pages 697–701. IEEE, 2007.

[14] S. Ha, S. Kim, C. Lee, Y. Yi, S. Kwon, and Y. Joo. Peace: A hardware-software codesign environment for multimedia embedded systems. *ACM Trans. on Design Automation of Elec. Sys.*, 12(3), 2007.

[15] 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.

[16] S. Li, N. Farahini, A. Hemani, K. Rosvall, and I. Sander. System level synthesis of hardware for dsp applications using pre-characterized function implementations. In *ACM/IEEE Int. Conf. on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2013.

[17] M.A. Kinsy and S. Devadas. Heracles 2.0: A tool for design space exploration of multi/many-core processors. In *Workshop on the Intersections of Computer Architecture and Reconfigurable Logic (CARL)*, june 2012.

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

[19] Eugene Cartwright, Azad Fahkari, Sen Ma, Christina Smith, Miaoqing Huang, D Andrews, and Jason Agron. Automating the design of mlut mpsopc fpgas in the cloud. In *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*, pages 231–236. IEEE, 2012.

[20] Blair Fort, Andrew Canis, Jongsok Choi, Nazanin Calagar, Ruolong Lian, Stefan Hadjis, Yu Ting Chen, Mathew Hall, Bain Syrowik, Tomasz Czajkowski, et al. Automating the design of processor/accelerator embedded systems with legup high-level synthesis. In *Embedded and Ubiquitous Computing (EUC), 2014 12th IEEE International Conference on*, pages 120–129. IEEE, 2014.

[21] MDE. Model-Based Engineering descritption. `http://modelbasedengineering.com`, 2015. Last accessed: 14/04/2015.

[22] Peter Feiler and David Gluch. *Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language*. Addison-Wesley Professional, 2012.

[23] 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.

[24] Opencores. Online OpenCores library. `http://opencores.org/`, 2014.

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

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

[27] 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.

[28] P. Coussy, C. Chavet, P. Bomel, D. Heller, E. Senn, and E. Martin. *High-Level Synthesis: From Algorithm to Digital Circuit*, chapter GAUT: A High-Level Synthesis Tool for DSP applications, pages 147–169. Springer, 2008.

[29] Y. Corre, J.P. Diguet, D. Heller, and L. Lagadec. A framework for high-level synthesis of heterogeneous mp-soc. In *Proceedings of the great lakes symposium on VLSI*, pages 283–286. ACM, 2012.

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

[31] 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.

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

[33] ATL. The Atlas Transformation Language (ATL). `http://www.eclipse.org/atl/`, 2014.

[34] Xtext. Xtext website. `https://eclipse.org/Xtext/index.html`, 2015. Last accessed: 10/04/2015.

[35] 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 Trans. on*, 24(12), 2005.

[36] 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.

[37] Xilinx. Xilinx XUPV5-LX110T FPGA Board Documentation. `http://www.xilinx.com/univ/xupv5-lx110t.htm`, 2011.

[38] Xilinx. Xilinx ML605 FPGA Board Documentation. `http://www.xilinx.com/products/boards/ml605/reference_designs.htm`, 2012.

[39] Altera. Stratix 10 - overview. `https://www.altera.com/products/fpga/stratix-series/stratix-10/overview.html`, 2015. Last accessed: 16/07/2015.